

C# - Módulo I

Carlos M P d Souza
77.351.980/0007-93



IMPACTA
EDITORIA

COD.: TE 1620/1_WEB

C# - Módulo I



IMPACTA
EDITORAS

Créditos

Copyright © TechnoEdition Editora Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da TechnoEdition Editora Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

C# - Módulo I

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Supervisão Editorial

Simone Rocha Araújo Pereira

Atualização

Carlos Magno Pratico de Souza

Revisão Ortográfica e Gramatical

Carolina A. Messias

Diagramação

Bruno de Oliveira Santos

Edição nº 1 | Cód.: 1620/1_WEB

outubro/ 2013

Sumário

Informações sobre o treinamento	10
Capítulo 1- Introdução ao Visual Studio e C#.....	11
1.1. Introdução	12
1.2. A linguagem C#	12
1.3. Programação orientada a objeto.....	13
1.4. Plataforma .NET	16
1.5. Conhecendo o Visual Studio.....	20
1.5.1. Requisitos para instalação.....	20
1.5.2. Página inicial.....	21
1.5.3. Iniciando um projeto.....	22
1.5.3.1. Janela ToolBox	25
1.5.3.2. Janela Properties	26
1.5.3.3. Solution Explorer	28
1.6. A classe Form1	30
1.6.1. Compilação.....	32
1.6.1.1. Compilando e executando o projeto.....	34
Pontos principais	36
Teste seus conhecimentos.....	37
Capítulo 2 - Formulários	41
2.1. Introdução	42
2.2. Criando uma interface.....	42
2.2.1. Formulário de inicialização.....	44
2.3. Conceitos importantes	46
2.3.1. Controles	46
2.3.2. Objetos	46
2.3.3. Propriedades.....	47
2.3.4. Métodos de evento.....	48
2.3.5. Métodos.....	50
2.4. Controles de formulário	51
2.5. Resumo dos principais controles e suas propriedades	53
2.5.1. Label e LinkLabel	53
2.5.1.1. Propriedades dos controles Label e LinkLabel.....	54
2.5.2. TextBox e RichTextBox	54
2.5.2.1. Propriedades dos controles TextBox e RichTextBox	55
2.5.2.2. Eventos do controle TextBox.....	56
2.5.3. Button.....	56
2.5.3.1. Propriedades do controle Button	57
2.5.3.2. Evento do controle Button	58
2.5.4. RadioButton	58
2.5.4.1. Propriedades do controle RadioButton	58
2.5.4.2. Eventos do controle RadioButton	59
2.5.5. CheckBox.....	60
2.5.5.1. Propriedades do controle CheckBox	60
2.5.5.2. Eventos do controle CheckBox	61

C# - Módulo I

2.5.6. ListBox.....	61
2.5.6.1. Propriedades do controle ListBox	62
2.5.6.2. Métodos do controle ListBox	63
2.5.7. ComboBox	63
2.5.7.1. Propriedades do controle ComboBox	64
2.5.8. DateTimePicker.....	64
2.5.8.1. Propriedades do controle DateTimePicker	64
2.5.9. TabControl.....	65
2.5.9.1. Propriedades do controle TabControl	65
2.5.10. Timer.....	66
2.5.10.1.Propriedades do controle Timer	66
2.6. Adicionando menus	66
2.6.1. MenuStrip	66
2.6.2. ToolStrip	68
2.6.2.1. Propriedades do controle MenuStrip.....	68
2.6.2.2. Propriedades do controle ToolStrip	68
2.6.3. Configurando as teclas de acesso para os comandos do menu.....	69
2.6.4. Convenções para criar menus	69
2.6.5. Executando as opções do menu	70
2.7. Adicionando barras de ferramentas.....	71
2.8. As caixas de diálogo padrão	72
Pontos principais	77
Teste seus conhecimentos.....	79
Mãos à obra!.....	83
 Capítulo 3 - Instruções, tipos de dados, variáveis e operadores	99
3.1. Introdução	100
3.2. Instruções	100
3.2.1. Identificadores	101
3.2.1.1. Palavras reservadas.....	101
3.3. Tipos de dados	102
3.4. Variáveis	103
3.4.1. Convenções	103
3.4.2. Declaração de variáveis	105
3.4.2.1. Variáveis locais de tipo implícito	105
3.5. Operadores	106
3.5.1. Operador de atribuição	107
3.5.2. Operadores aritméticos.....	108
3.5.3. Operadores relacionais	110
3.5.4. Operadores lógicos	111
3.5.5. Operador ternário	113
Pontos principais	114
Teste seus conhecimentos.....	115
Mãos à obra!.....	119

Sumário

Capítulo 4 - Instruções de decisão	125
4.1. Introdução	126
4.2. Instruções de decisão if / else e switch / case	126
4.2.1. If / else	126
4.2.2. Switch / case.....	131
Pontos principais	134
Teste seus conhecimentos.....	135
Mãos à obra!	141
Capítulo 5 - Instruções de repetição.....	147
5.1. Introdução	148
5.2. Instruções de repetição ou iteração.....	148
5.2.1. While	148
5.2.2. Do / while	150
5.2.3. For.....	151
5.2.4. Break	157
5.2.5. Continue.....	159
Pontos principais	161
Teste seus conhecimentos.....	163
Mãos à obra!	167
Capítulo 6 - Arrays.....	171
6.1. O que são arrays?.....	172
6.2. Construção e instanciação de arrays	172
6.3. Conhecendo o tamanho de um array.....	175
6.3.1. Arrays com várias dimensões	176
6.4. Passando um array como parâmetro	177
6.4.1. Palavra-chave params	178
6.4.1.1. Params object[]	179
6.5. Exemplos	180
6.5.1. Exemplo 1	180
6.5.2. Exemplo 2	185
6.5.3. Exemplo 3	188
6.5.4. Exemplo 4	189
Pontos principais	194
Teste seus conhecimentos.....	195
Mãos à obra!	201
Capítulo 7 - Manipulando arquivos texto.....	205
7.1. Classes StreamWriter e StreamReader.....	206
7.1.1. StreamWriter	206
7.1.2. StreamReader.....	207
7.2. Exemplo	207
Pontos principais	226
Teste seus conhecimentos.....	227

C# - Módulo I

Capítulo 8 - Enumerações e Delegates	231
8.1. Enumerações	232
8.1.1. Exemplo	234
8.2. Delegates.....	242
Pontos principais	248
Teste seus conhecimentos.....	249
Mãos à obra!.....	253
Capítulo 9 - Classes e estruturas	263
9.1. Introdução	264
9.2. Semelhanças entre classes e estruturas.....	269
9.3. Diferenças entre classes e estruturas	271
9.4. Classes	272
9.4.1. Exemplo: Criação de propriedades, métodos e construtores.....	272
9.4.2. Exemplo: Classe Circulo	280
9.4.3. Exemplo: Classe ConvTemperaturas.....	282
9.4.4. Exemplo: Classe CalcMedia	284
9.5. Classes static.....	286
9.5.1. Modificador static	289
9.5.1.1. Membros estáticos	289
9.5.2. Exemplo: Classe Utils	290
9.6. Métodos de extensão	294
9.6.1. Exemplo: Classe MetodosExtensao.....	295
9.7. Herança	301
9.7.1. Criando uma herança	301
9.7.2. Acesso aos membros da classe pai por meio do operador base.....	304
9.7.3. Métodos sobrecarregados (overloaded methods).....	305
9.7.4. Polimorfismo	307
9.7.5. Palavras-chaves virtual e override	307
9.7.6. Exemplo: Class Library LibComponentes	308
9.7.7. Componentes visuais ou controles	314
9.7.8. Exemplo: Herdeiros de Control	316
9.7.9. Exemplo: Herdeiros de controles já existentes	327
Pontos principais	333
Teste seus conhecimentos.....	335
Mãos à obra!.....	339
Capítulo 10 - Expressões regulares	341
10.1. Introdução	342
10.2. Principais símbolos usados em uma expressão regular	342
10.3. A classe Regex	344
10.4. Exemplo: Projeto para testar expressões regulares.....	345
Pontos principais	349
Teste seus conhecimentos.....	351

Sumário

Capítulo 11 - Coleções	355
11.1. Introdução	356
11.2. Diferenças entre coleções e arrays	357
11.3. ArrayList	357
11.4. Classe Stack	361
11.5. Classe Queue	364
11.6. List<tipo>	366
11.7. Inicializadores de List	369
Pontos principais	370
Teste seus conhecimentos.....	371
Capítulo 12 - Acesso à API do Windows.....	375
12.1. Introdução	376
12.2. Exemplo: WindowsDLL	376
12.2.1. Arredondando as bordas de um formulário	377
12.2.2. Utilizando algumas funções de API.....	379
Pontos principais	386
Capítulo 13 - Gerando Setup de instalação	387
13.1. Introdução	388
13.2. Instalação do InstallShield LE	388
13.3. Usando o InstallShield	394
Pontos principais	401

Informações sobre o treinamento

Para que os alunos possam obter um bom aproveitamento do **curso de C# - Módulo I**, é imprescindível que eles tenham participado dos nosso curso de Introdução à Lógica de Programação, ou possuam conhecimentos equivalentes.

Introdução ao Visual Studio e C#

1

- ✓ A linguagem C#;
- ✓ Programação orientada a objeto;
- ✓ Plataforma .NET;
- ✓ Conhecendo o Visual Studio;
- ✓ A classe Form1.



IMPACTA
EDITORA

1.1. Introdução

A linguagem de programação C# é uma das linguagens disponíveis na plataforma .NET. Em conjunto com o Visual Studio, que é a ferramenta utilizada para desenvolver aplicações usando C#, podemos criar diversos tipos de aplicação, por exemplo, bibliotecas de classe (.DLL), páginas WEB, aplicações desktop, etc.

Neste capítulo inicial, abordaremos assuntos básicos, que servirão de fundamento para o trabalho com o C#: a programação orientada a objeto, a plataforma .NET, o Visual Studio, a introdução a uma programação elementar e os conceitos básicos sobre compilação e execução de um programa.

1.2. A linguagem C#

A linguagem C# é utilizada para a criação de vários tipos de aplicativos, como de banco de dados, cliente-servidor, aplicativos tradicionais do Windows, além de componentes distribuídos, XML Web Services, entre outros. Trata-se de uma linguagem orientada a objeto, com tipagem segura, cuja implementação em sistemas Microsoft é o Visual C#. A edição 2012 conta com um depurador integrado, um editor de códigos avançado, designers de interface de usuários altamente funcionais, entre outras ferramentas que auxiliam na criação de aplicativos a serem rodados no .Net Framework, versão 4.5.

Embora seja fácil de usar, a linguagem C# é bastante sofisticada. Usuários que tenham conceitos básicos de C, C++ ou Java reconhecerão rapidamente detalhes de seu funcionamento e sintaxe.



O termo “tipagem” significa que todos os dados utilizados na programação precisam ter um tipo definido.

1.3. Programação orientada a objeto

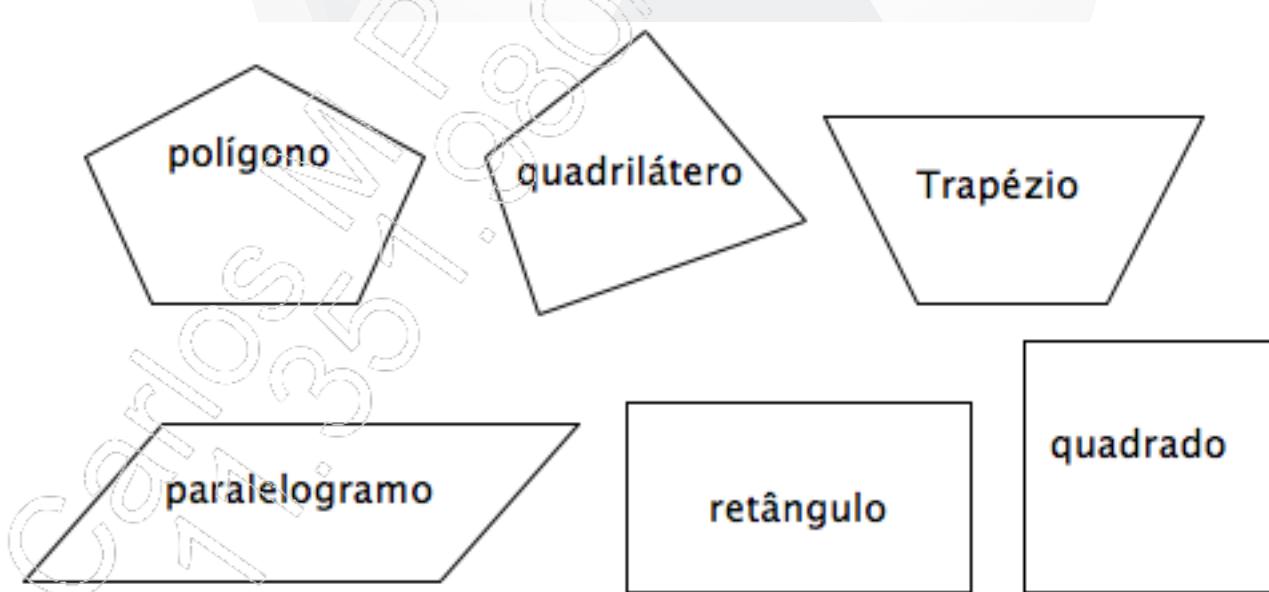
Este conceito de programação, que vem sendo utilizado desde a década de 1960, consiste em aplicar aos módulos de programação as mesmas características existentes nos objetos que utilizamos no nosso dia a dia.

- **Classe de objeto:** Corresponde à categoria do objeto em questão. Quando alguém diz a palavra automóvel, todos sabem do que se trata, mesmo sem ter um automóvel real em sua frente. Este automóvel que as pessoas visualizam corresponde à classe à qual pertencem todos os objetos que podemos ver no mundo real:
 - A classe é uma entidade teórica, ou seja, ela não existe fisicamente e não ocupa lugar na memória;
 - Quando criamos um objeto de uma determinada classe (instanciamos), o objeto criado passa a existir e a ocupar espaço na memória;
 - A classe é apenas a ideia, é a definição de todas as características que um objeto pertencente a ela pode assumir;
 - Em programação, poderíamos criar uma classe chamada **CalcImpostosPessoaFisica**, que seria responsável por todos os cálculos de impostos para pessoa física.
- **Propriedades:** Para poder definir e controlar todas as características que os objetos podem ter, a classe as armazena em propriedades:
 - Para o automóvel, por exemplo, poderíamos destacar as propriedades: Cor, Potência do motor, Quantidade de passageiros, Tipo de combustível etc;
 - Para **CalcImpostosPessoaFisica**, podemos ter as propriedades: Nome, Salário Bruto, Quantidade de Dependentes e quaisquer outras informações envolvidas no procedimento de cálculo de impostos.

C# - Módulo I

- **Métodos:** São ações que a classe executa a nosso pedido:
 - No caso do automóvel: Acelerar, frear, trocar a marcha, etc;
 - No caso de **CalcImpostosPessoaFisica**: Calcular o IRPF, Calcular o INSS, Imprimir resultados.
- **Eventos:** São ações externas que a classe consegue detectar. No entanto, não há nenhuma reação automática quando elas ocorrem, ficando a critério de quem instanciou a classe criar ou não uma reação a essas ações externas:
 - O automóvel permite que o seu condutor veja tudo o que ocorre do lado de fora, mas, quando o semáforo fica vermelho, por exemplo, o automóvel não para automaticamente. Cabe ao condutor do veículo enxergar este evento e reagir a ele;
 - Um evento é muito semelhante a uma propriedade, só que em vez de armazenar um dado, ele armazena um método, uma ação.

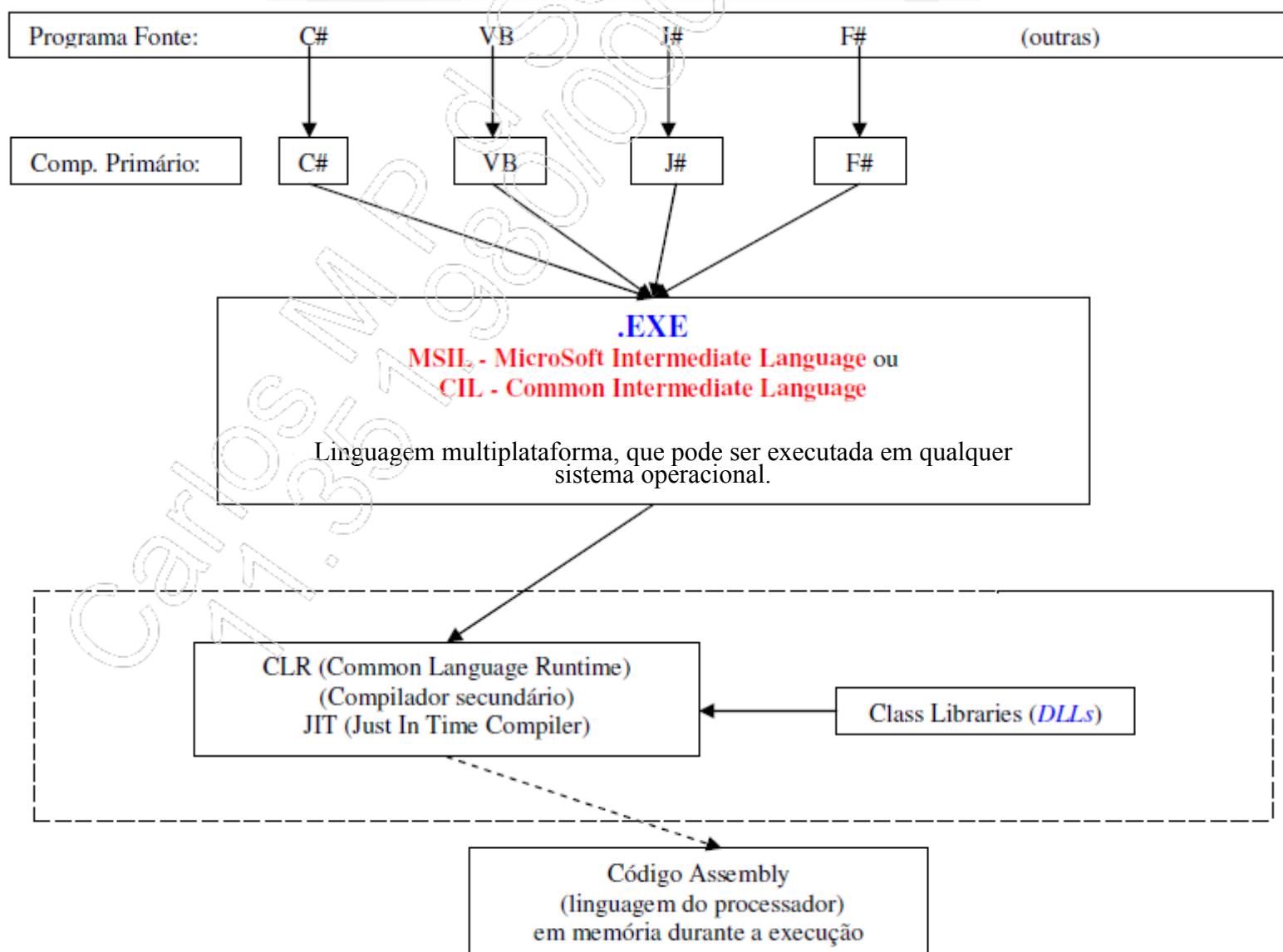
- **Herança:** Uma classe pode ser “filha” ou herdeira de uma já existente. Neste caso, ela herda todas as propriedades, métodos e eventos da classe “mãe” (ancestral). Em C#, não existe herança múltipla, o ancestral direto pode ser apenas uma classe. Além de tudo que a classe herdeira recebeu dos seus ancestrais (mãe, avó, bisavó, etc.), ela pode também criar novas propriedades, métodos e eventos. Por exemplo:
 - Polígono: Figura plana formada pela ligação de pelo menos três pontos com retas;
 - Quadrilátero: É um polígono formado pela ligação de quatro pontos;
 - Trapézio: É um quadrilátero que possui dois lados paralelos;
 - Paralelogramo: É um trapézio que possui quatro lados paralelos, dois a dois;
 - Retângulo: É um paralelogramo cujo ângulo formado entre dois lados adjacentes é de 90 graus;
 - Quadrado: É um retângulo cujas medidas dos quatro lados são iguais.



1.4. Plataforma .NET

De forma simplificada, a plataforma .NET é composta pelos seguintes elementos:

- **Visual Studio**: Ferramenta para desenvolvimento de aplicações. Ele contém o compilador primário que gera o executável;
- **Compilador primário**: Transforma o programa que fizemos em linguagem de alto nível (C#, C++, VB.Net) em uma outra linguagem (MSIL) e gera um arquivo .EXE;
- **Bibliotecas de classes**: Conjunto de DLLs contendo milhares de recursos disponíveis para utilizarmos nas nossas aplicações;
- **Compilador secundário**: Lê as instruções contidas no arquivo .EXE e as traduz para a linguagem do microprocessador (assembly).



- Trecho de código em C#:

```
private void btnCalcula_Click(object sender, EventArgs e)
{
    // declarar uma variável para cada dado envolvido no processo
    int numero, quadrado;
    try
    {
        // receber os dados de entrada
        numero = Convert.ToInt32(tbxNumero.Text);
        // se o número for maior que a raiz quadrada de 2000000000...
        if (numero > Math.Sqrt(2000000000))
        {
            // exibir mensagem
            MessageBox.Show("Número excede o limite...");
            tbxNumero.Focus(); // volta o foco para o TextBox
            tbxNumero.SelectAll(); // seleciona tudo
            // finalizar a execução deste método
            return;
        } // fim do IF
        // calcular os dados de saída
        quadrado = numero * numero;
        // mostrar o resultado na tela
        lblQuadrado.Text = quadrado.ToString();
    } // fecha o TRY
    catch (FormatException)
    {
        MessageBox.Show("O dado não é numérico...");
    }
    catch (OverflowException)
    {
        MessageBox.Show("O valor excede o limite de inteiro...");
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}
```

C# - Módulo I

- **Código em MSIL/CIL (dentro do .EXE gerado pelo compilador primário):**

```
.method private hidebysig instance void btnCalcula_Click(object
    sender,
    class [mscorlib]System.EventArgs e) cil managed
{
// Code size 114 (0x72)
    .maxstack 2
    .locals init ([0] int32 numero,
    [1] int32 quadrado,
    [2] class [mscorlib]System.Exception ex,
    [3] bool CS$4$0000)
    IL_0000: nop
    .try
    {
        IL_0001: nop
        IL_0002: ldarg.0
        IL_0003: ldfld class [System.Windows.Forms]System.Windows.Forms.
            TextBox Quadrado.Form1::tbxNumero
        IL_0008: callvirt instance string [System.Windows.Forms]System.
            Windows.Forms.Control::get_Text()
        IL_000d: call int32 [mscorlib]System.Convert::ToInt32(string)
        IL_0012: stloc.0
        IL_0013: ldloc.0
        IL_0014: conv.r8
        IL_0015: ldc.r8 2000000000.
        IL_001e: call float64 [mscorlib]System.Math::Sqrt(float64)
        IL_0023: cgt
        IL_0025: ldc.i4.0
        IL_0026: ceq
        IL_0028: stloc.3
        IL_0029: ldloc.3
        IL_002a: brtrue.s IL_003a
        IL_002c: nop
        IL_002d: ldstr bytearray (4E 00 FA 00 6D 00 65 00 72 00 6F 00 20 00
            65 00 // N...m.e.r.o. .e.
            78 00 63 00 65 00 64 00 65 00 20 00 6F 00 20 00 // x.c.e.d.e. .o. .
            6C 00 69 00 6D 00 69 00 74 00 65 00 2E 00 2E 00 // l.i.m.i.t.e.....
            2E 00 ) // ..
        IL_0032: call valuetype [System.Windows.Forms]System.Windows.Forms.
            DialogResult
```

```
[System.Windows.Forms]System.Windows.MessageBox::Show(string)
IL_0037: pop
IL_0038: leave.s IL_0070
IL_003a: ldloc.0
IL_003b: ldloc.0
IL_003c: mul
IL_003d: stloc.1
IL_003e: ldarg.0
IL_003f: ldfld class [System.Windows.Forms]System.Windows.Forms.Label
Quadrado.Form1::lblQuadrado
IL_0044: ldloca.s quadrado
IL_0046: call instance string [mscorlib]System.Int32::ToString()
IL_004b: callvirt instance void [System.Windows.Forms]System.Windows.
Forms.Control::set_Text(string)
IL_0050: nop
IL_0051: nop
IL_0052: leave.s IL_006f
} // end .try
catch [mscorlib]System.Exception
{
IL_0054: stloc.2
IL_0055: nop
IL_0056: ldstr bytearray (44 00 61 00 64 00 6F 00 20 00 6E 00 E3 00
6F 00 // D.a.d.o. .n....o.
20 00 E9 00 20 00 6E 00 75 00 6D 00 E9 00 72 00 // ... .n.u.m....r.
69 00 63 00 6F 00 2E 00 2E 00 2E 00 ) // i.c.o.....
IL_005b: ldloc.2
IL_005c: callvirt instance string [mscorlib]System.Object::ToString()
IL_0061: call string [mscorlib]System.String::Concat(string,
string)
IL_0066: call valuetype [System.Windows.Forms]System.Windows.Forms.
DialogResult
[System.Windows.Forms]System.Windows.MessageBox::Show(string)
IL_006b: pop
IL_006c: nop
IL_006d: leave.s IL_006f
} // end handler
IL_006f: nop
IL_0070: nop
IL_0071: ret} // end of method Form1::btnCalcula_Click
```

1.5. Conhecendo o Visual Studio

O Integrated Development Environment (IDE) do Visual Studio é um ambiente de desenvolvimento utilizado para escrever programas. Por se tratar de um software de programação potente e personalizável, o Visual Studio possui todas as ferramentas necessárias para desenvolver programas sólidos para o Windows e para a Web de forma rápida e eficiente. Muitas das características encontradas no Visual Studio IDE se aplicam de forma idêntica às características do Visual Basic, Visual C++, Visual J# e Visual C#.

Como trataremos especificamente do C#, devemos ressaltar que o Visual Studio possui todas as funcionalidades e ferramentas necessárias à criação de projetos nessa linguagem. Ainda que o ambiente em si não seja Visual C#, a linguagem utilizada é Visual C#.

1.5.1. Requisitos para instalação

Os requisitos de sistema para a instalação do Visual Studio 2012 são os seguintes:

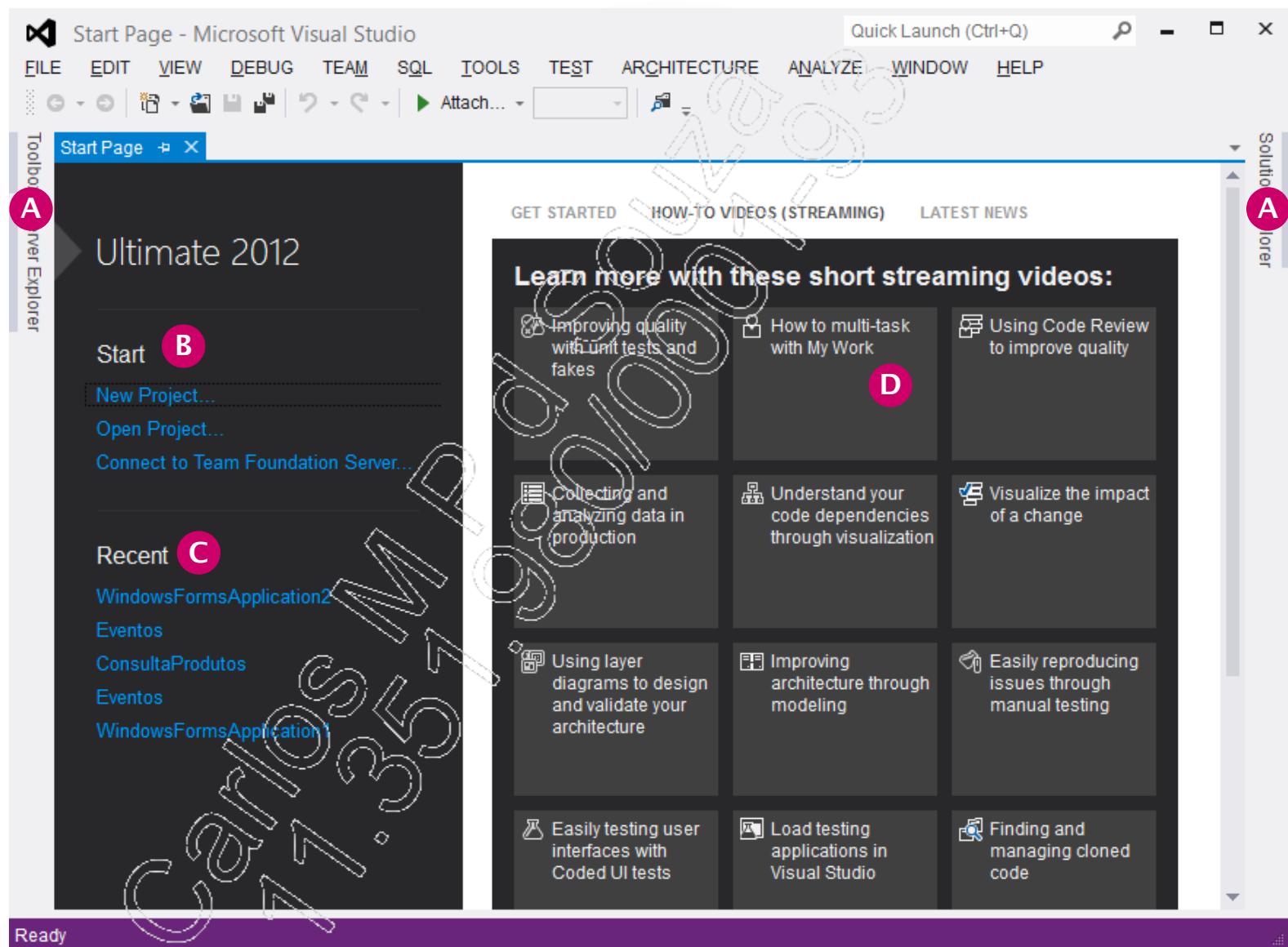
- Windows 7 SP1 (x86 e x64);
- Windows 8 (x86 e x64);
- Windows Server 2008 R2 (x64);
- Windows Server 2012 (x64).

As arquiteturas compatíveis são as de 32 bits (x86) e de 64 bits (x64). Como requisitos de hardware, temos:

- Processador de 1.6GHz ou superior;
- 1 GB de RAM;
- 10 GB disponível em disco;
- Unidade de disco rígido de 5400 RPM;
- Placa de vídeo compatível com DirectX 9, com resolução de vídeo de 1280 x 1024 ou superior;
- Unidade de DVD-ROM.

1.5.2. Página inicial

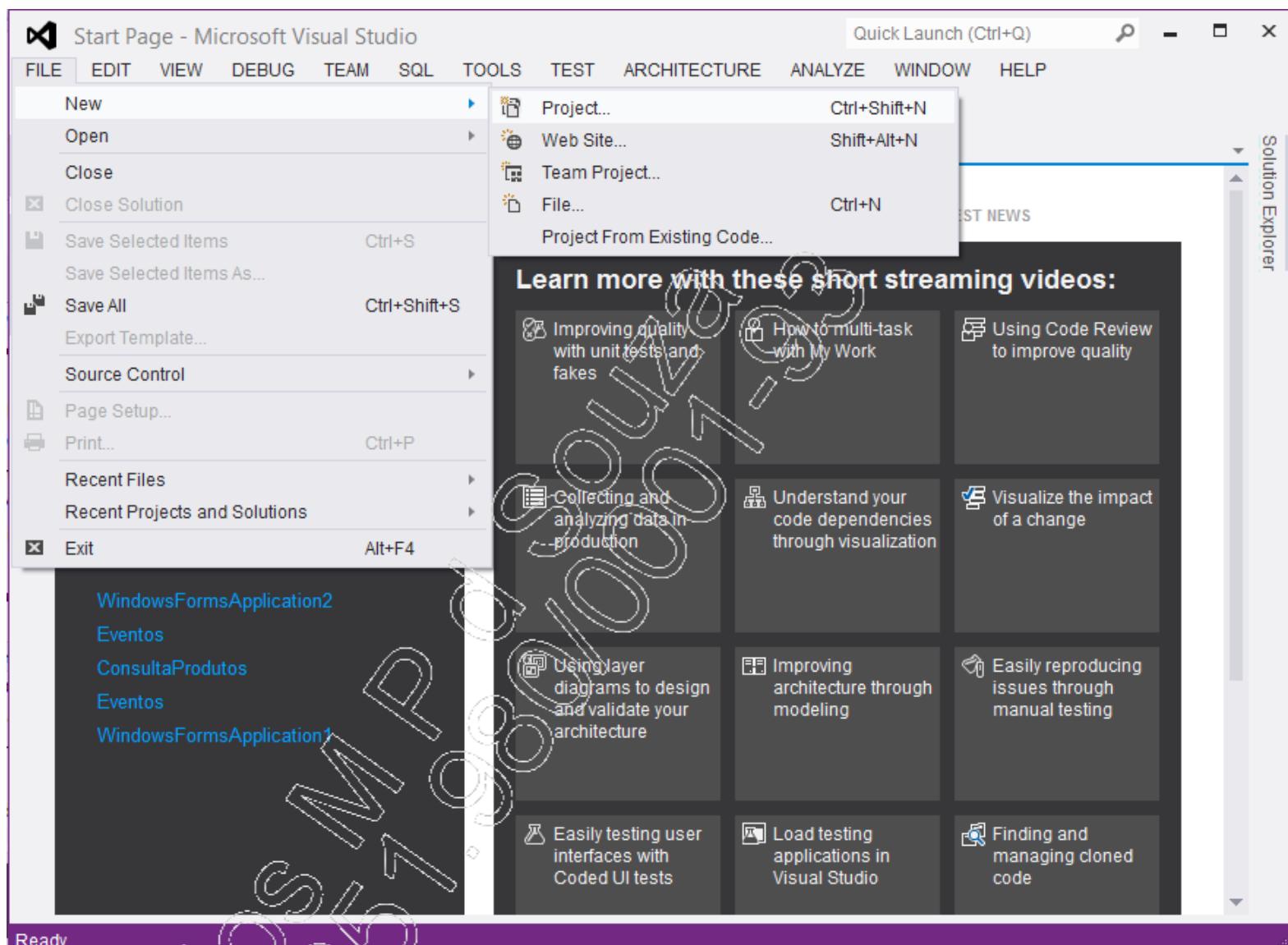
A tela inicial do Visual Studio 2012 possui uma barra de menu na parte superior e três janelas muito importantes que, na imagem a seguir, estão retraídas nas laterais esquerda e direita da tela principal (A). Na área central da janela (B), podemos ver as opções **New Project** (Novo Projeto), **Open Project** (Abrir Projeto), além dos últimos projetos que foram abertos (C) e vários vídeos (D) sobre o Visual Studio e a plataforma .Net.



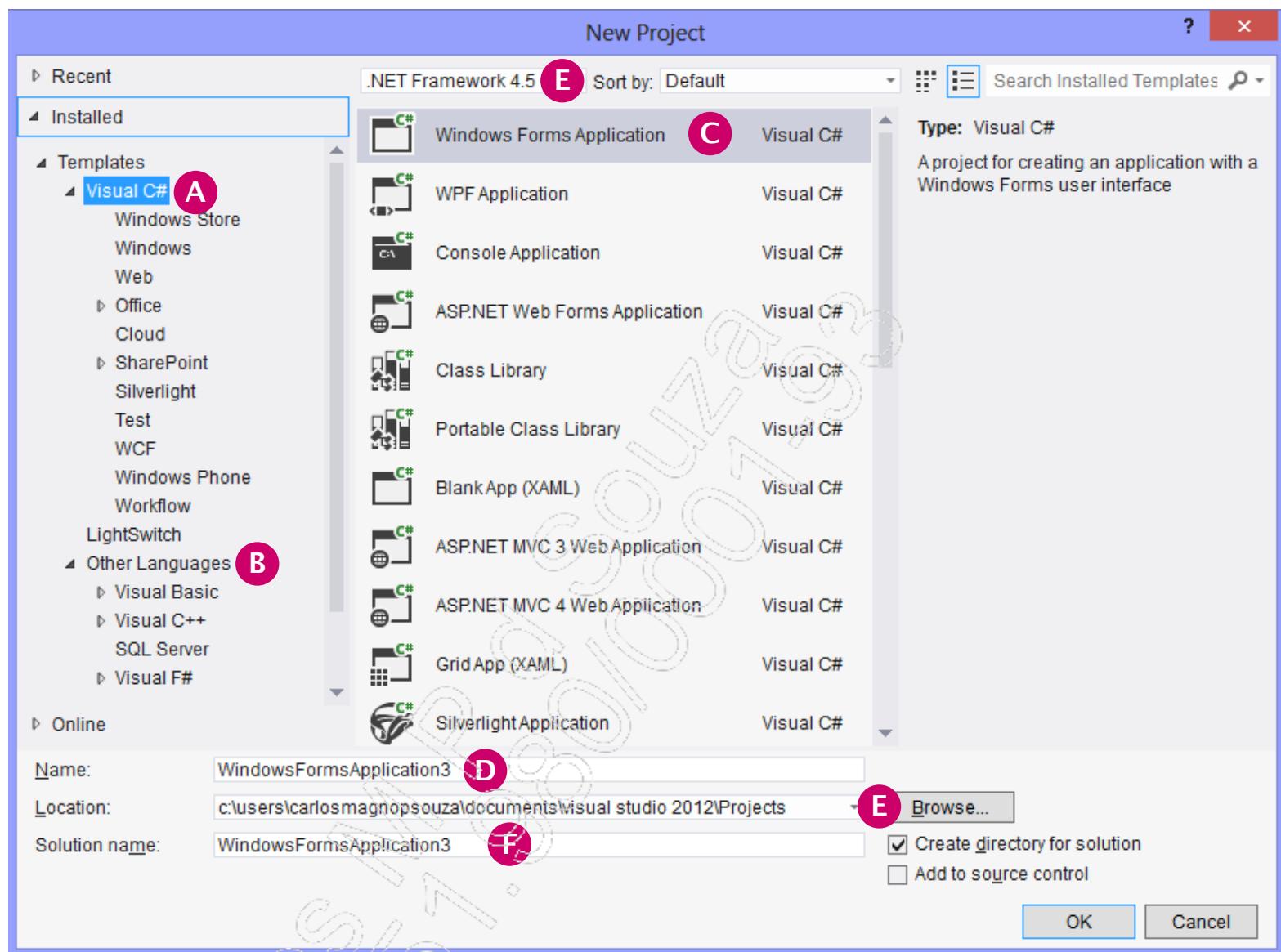
Para detalhar melhor as janelas indicadas, precisamos criar um projeto.

1.5.3. Iniciando um projeto

Para iniciar um projeto, basta clicar sobre o botão **New Project** ou, no menu **File**, selecionar **New / Project**.



Então, será aberta a caixa de diálogo **New Project**, que lista, de acordo com a linguagem de programação utilizada e o tipo de aplicativo, os templates instalados, os quais podem ser usados como base para a criação de um projeto. Vejamos alguns detalhes da caixa de diálogo **New Project**:



- A - Linguagem de programação que será usada para desenvolver o projeto;
- B - Outras linguagens disponíveis;
- C - Tipo de projeto (aplicação) que será desenvolvido;
- D - Nome do projeto;
- E - Diretório em que o projeto será salvo. Use o botão **Browse** para alterar o diretório;
- F - Nome da solução (Solution). Uma solution pode conter vários projetos ao mesmo tempo.

C# - Módulo I

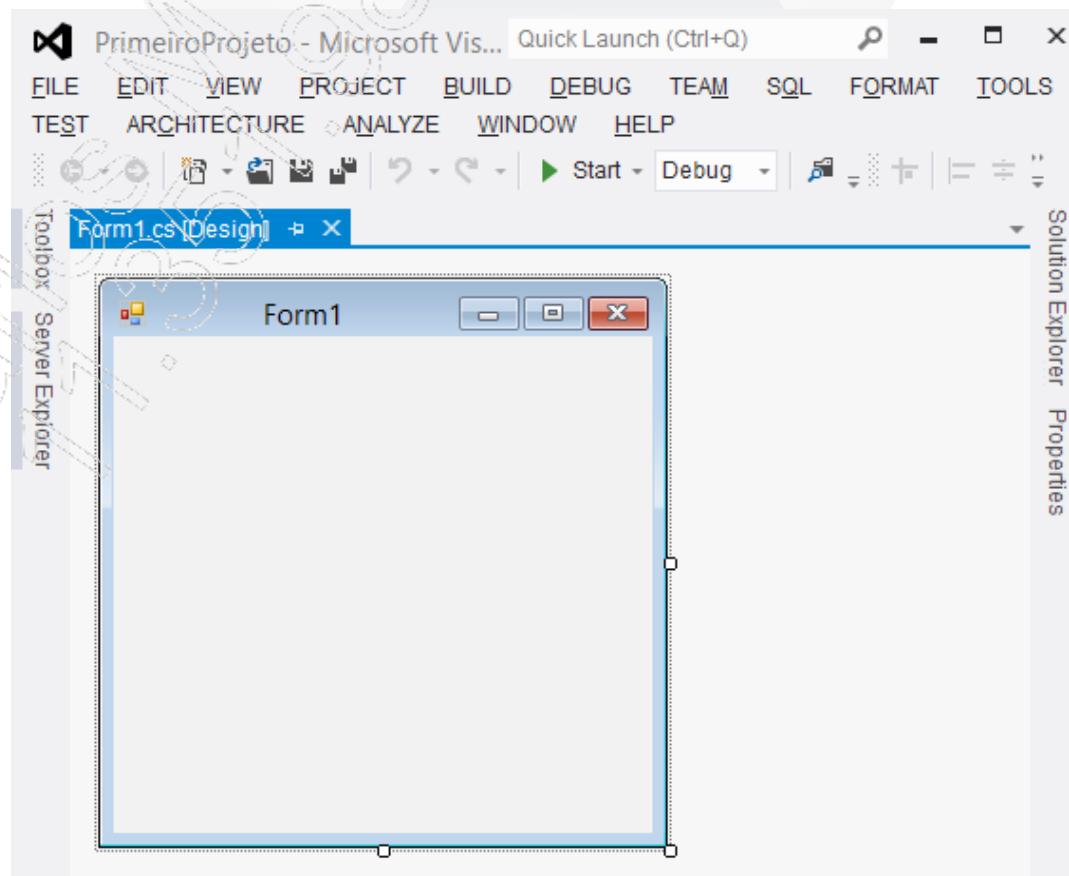
O nome do projeto não pode conter nenhum dos caracteres a seguir:

#	Cerquilha	/	Barra
%	Percentual	\	Barra invertida
&	E comercial	:	Dois pontos
*	Asterisco	"	Aspas duplas
?	Ponto de interrogação	<	Menor que
	Barra vertical	>	Maior que
	Espaço inicial ou final		

Mantenha selecionada a linguagem C# e a aplicação **Windows Form**. Altere a parte inferior de acordo com a imagem a seguir. Em seguida, clique em **OK**:

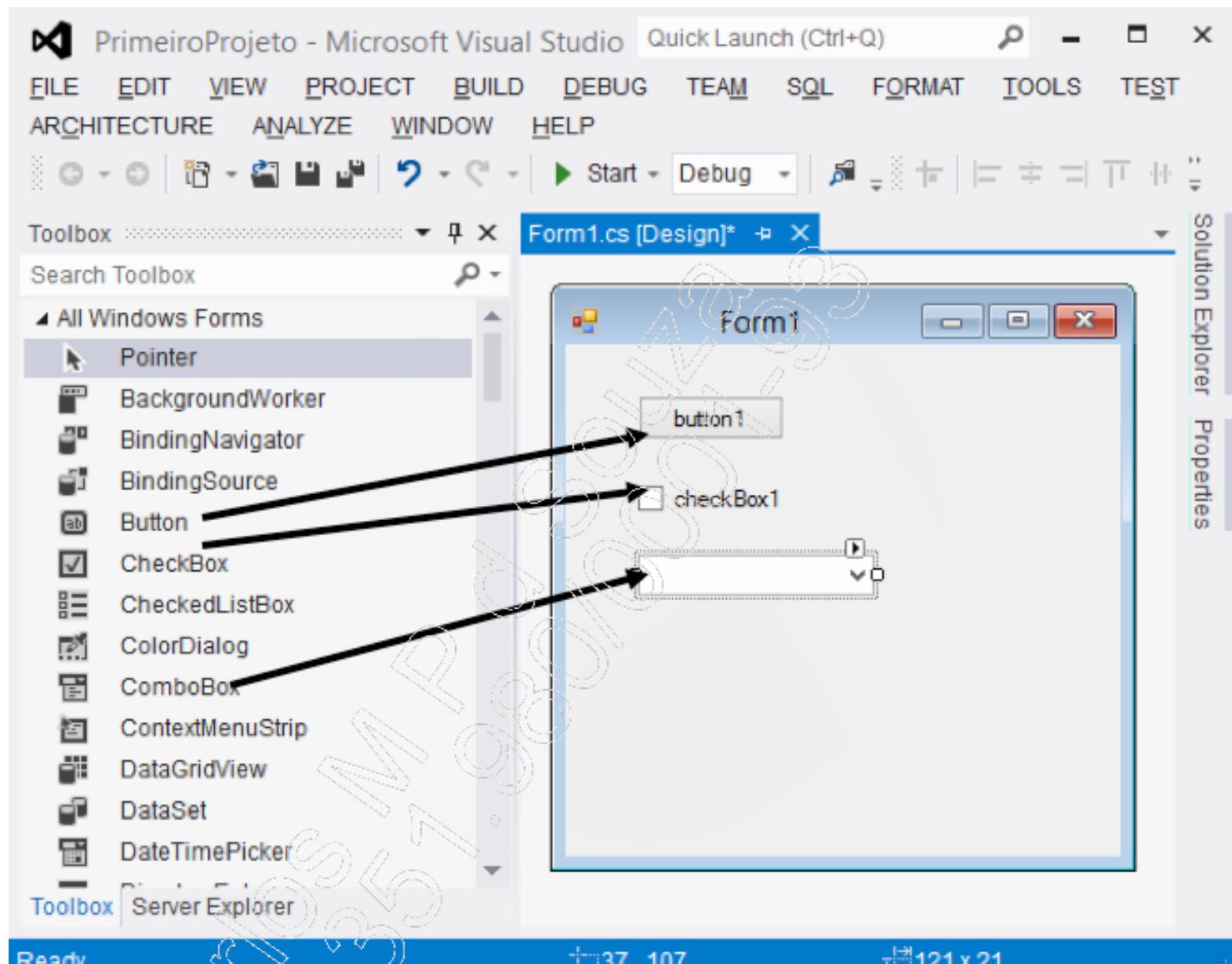


A janela identificada como **Form1** que aparece na parte central corresponde à tela principal do nosso projeto. A partir dela, outras poderão ser abertas.



1.5.3.1.Janela ToolBox

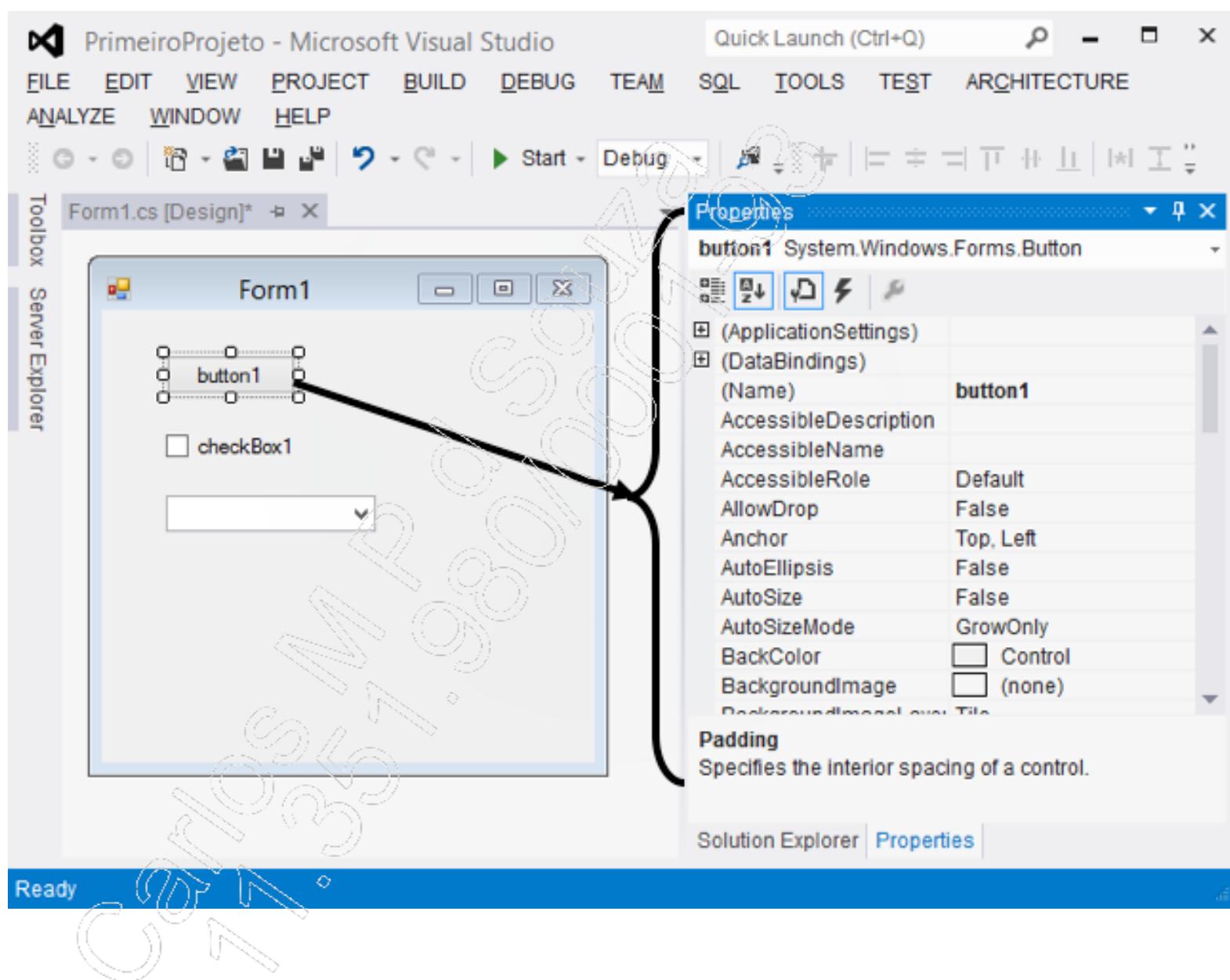
Esta janela possui os componentes necessários para a montagem do nosso formulário. Para isso, basta arrastar os elementos da **ToolBox** para a **Form**.



1.5.3.2.Janela Properties

Essa janela permite verificar e alterar as propriedades do objeto selecionado, além de criar eventos para ele.

A janela **Properties**, exibida a seguir, possui diversos tipos de campos de edição, em função das necessidades de cada propriedade. As propriedades exibidas em cinza estão disponíveis apenas para leitura.

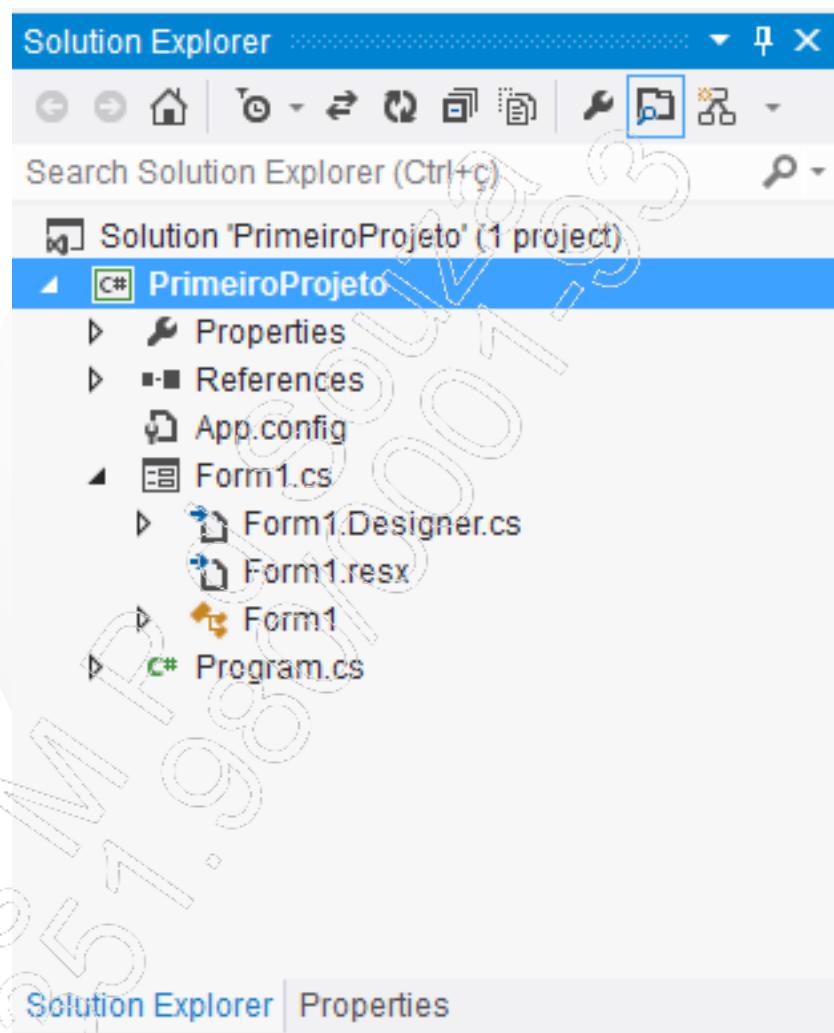


A tabela a seguir apresenta uma breve descrição dos elementos que compõem a janela **Properties**:

Elemento	Ícone / Caixa	Descrição
Nome do objeto	<code>button1 System.Windows.Forms.Button</code>	Nome do objeto selecionado e a classe a qual ele pertence.
Categorized		Organiza as propriedades do objeto selecionado de acordo com a categoria.
Alphabetical		Classifica as propriedades e eventos em ordem alfabética.
Properties		Exibe as propriedades de um objeto.
Events		Exibe os eventos de um objeto. Disponível apenas quando um formulário ou controle de formulário estiver ativo.
Property Pages		Exibe a caixa de diálogo Property Pages , que mostra os conjuntos ou subconjuntos de propriedades do item selecionado exibidos na janela Properties .
Description panel	 Padding Specifies the interior spacing of a control.	Mostra o nome da propriedade e uma breve descrição sobre ela.

1.5.3.3.Solution Explorer

A janela **Solution Explorer** fornece uma visualização estrutural dos projetos, arquivos e quaisquer itens que compõem uma solução. Eles são exibidos em uma estrutura hierárquica de pastas, possibilitando, dessa forma, fácil gerenciamento das tarefas e acesso rápido aos itens, para qualquer alteração necessária. Para acessar esta janela, ilustrada a seguir, basta selecionar **Solution Explorer** no menu **View**.



Vamos conhecer, a seguir, algumas opções dessa janela:

- **Nome da Solução** Solution 'PrimeiroProjeto' (1 project) : Uma solução pode armazenar vários projetos. Neste caso, esta solução armazena um único projeto:
 - **Nome do Projeto** PrimeiroProjeto
- **Propriedades** Properties: Armazena configurações, imagens e outros elementos do projeto;
- **Referências** References: Armazena as bibliotecas necessárias para que o nosso projeto possa ser compilado. Uma biblioteca contém um conjunto de classes e tipos de dados que o nosso projeto utiliza. A classe **Button**, à qual pertence o objeto **button1** que colocamos no nosso formulário, foi criada dentro da biblioteca **System.Windows.Forms**:
 - **Formulário** Form1.cs : Arquivos que compõem o nosso formulário;
 - Form1.cs
 - Form1.Designer.cs
 - Form1.resx
 - Form1
 - **Módulo de partida do projeto** Program.cs : É este módulo do projeto que coloca o formulário principal do projeto em funcionamento.

1.6. A classe Form1

A partir do Solution Explorer, podemos clicar com o botão direito do mouse sobre **Form1.cs** e selecionar a opção **View Code** para ver o código C# referente ao formulário. Também podemos visualizar esse código pressionando a tecla F7, que alterna o design do formulário com o código C#.

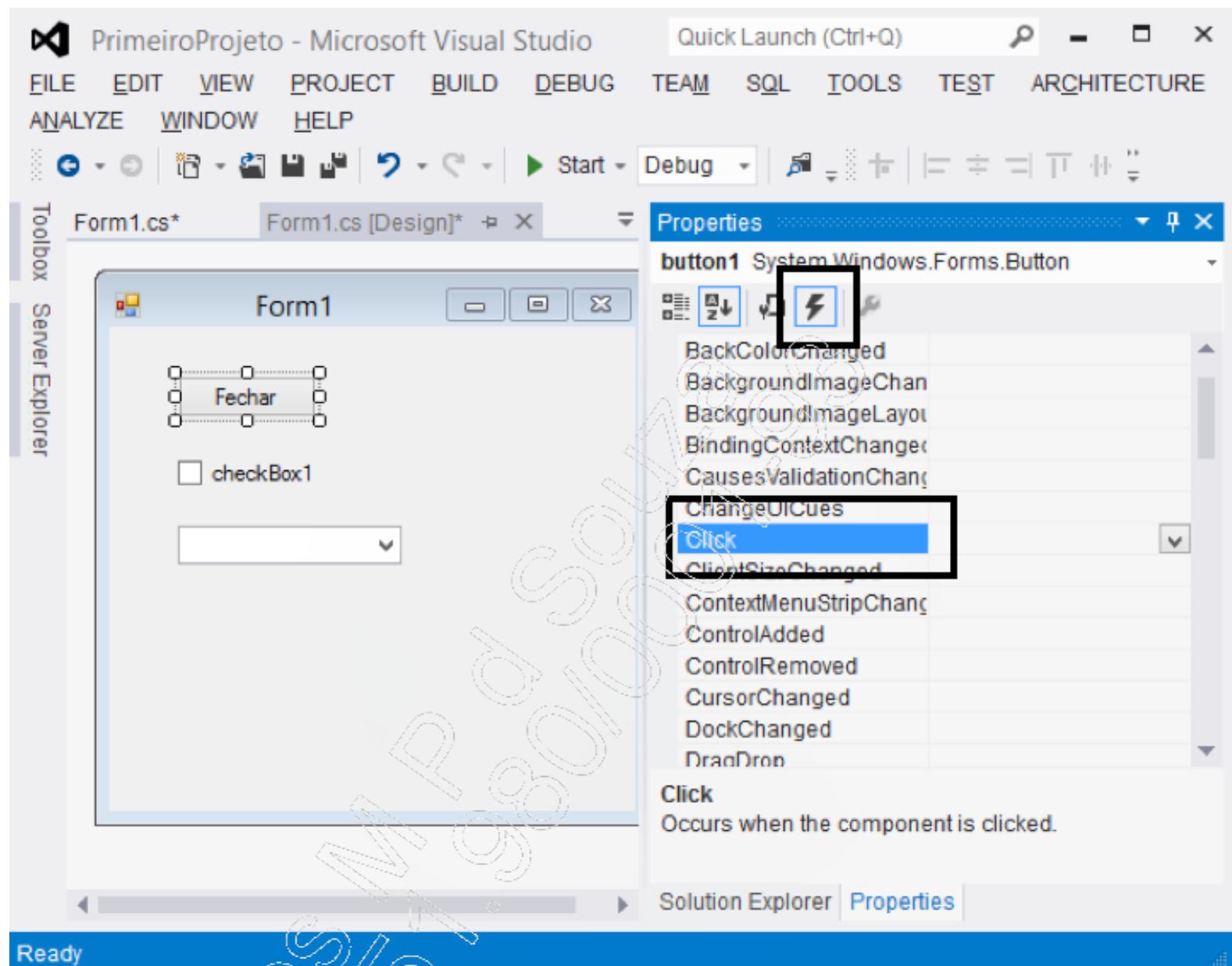
Um formulário é composto basicamente por dois arquivos: **Form1.cs** e **Form1.Designer.cs**.

- **Form1.cs**: É onde programamos todas as ações que o formulário vai executar;
- **Form1.Designer.cs**: É responsável pela criação visual do formulário. Sempre que arrastamos um objeto da ToolBox para o formulário, o Visual Studio acrescenta o código em **Form1.Designer.cs**. Quando arrastamos o objeto **Button** para o **Form**, o VS2012 cria o seguinte código:

```
// cria (instancia) um objeto da classe Button
this.button1 = new System.Windows.Forms.Button();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(100, 106);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "Fechar";
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(292, 273);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);

private System.Windows.Forms.Button button1;
```

Selecione o objeto **Button** e vá até a janela de propriedades. Então, altere a propriedade **Text** por **Fechar**. Em seguida, selecione **Eventos**, localize o evento **Click** e aplique um duplo-clique sobre ele.



1.6.1. Compilação

Vejamos as indicações no código a seguir:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq; A
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace PrimeiroProjeto B
{
    public partial class Form1 : Form C
    {
        public Form1() D
        {
            InitializeComponent();
        }

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

- A - Bibliotecas ou namespaces que o formulário vai usar. Quando vamos acessar as propriedades ou métodos de um objeto que está em uma biblioteca, devemos escrever o nome dela antes. Se fôssemos escrever uma linha de código para alterar o tipo de letra do objeto **button1**, deveríamos utilizar a classe **Font**, que foi definida dentro de **System.Drawing**. O código então ficaria assim:

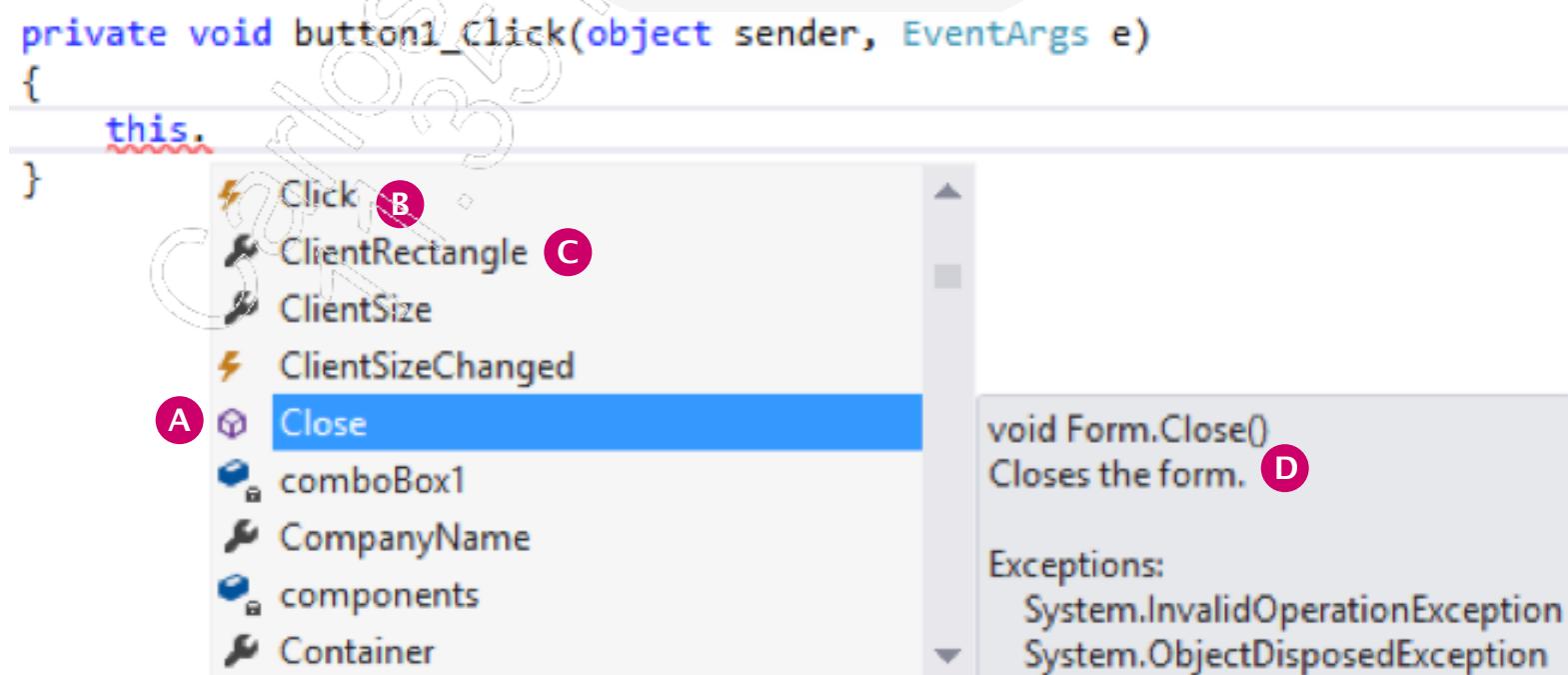
```
button1.Font = new System.Drawing.Font("Arial", 12);
```

Agora, se referenciamos a biblioteca no início do arquivo, podemos omitir o nome do namespace antes da classe **Font**:

```
using System.Drawing;
...
...
button1.Font = new Font("Arial", 12);
```

- **B** - Namespace do projeto atual. Todo projeto tem um namespace e, normalmente, todos os arquivos do projeto estão contidos no mesmo namespace;
- **C** - Definição da classe **Form1**:
 - **public**: Indica que a classe poderá ser usada neste e em outros projetos;
 - **partial**: Indica que a classe está separada em dois ou mais arquivos. No caso do **Form1**, a classe está separada em **Form1.cs** e **Form1.Designer.cs**;
 - **class Form1**: Nome da classe;
 - **:Form**: Indica que a classe **Form1** é herdeira da classe **Form**.
- **D** - Método construtor da classe **Form1**. Toda classe possui um método construtor, que é responsável por criar o objeto;
- **E** - Método que será executado quando o objeto **button1** for clicado.

Dentro do método **button1_Click**, digite **this.Close()**, que é o método que fecha o formulário. Assim que escrever **this.**, o VS2012 usará o IntelliSense para abrir o seguinte quadro de opções:

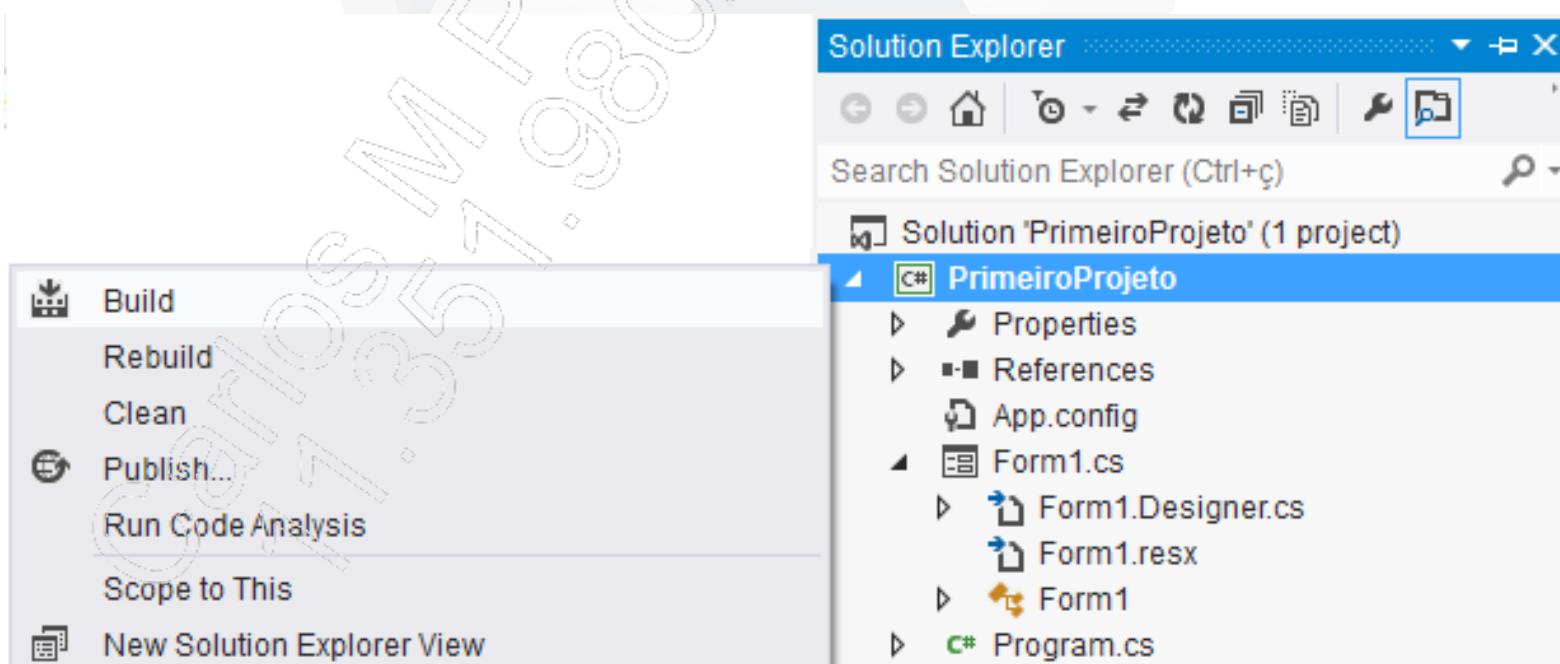


- **A** – Método;
- **B** – Evento;
- **C** – Propriedade;
- **D** – Explicação sobre o item selecionado.

O C# é sensível às letras minúsculas e maiúsculas, portanto **Close** deve ser escrito exatamente como mostrado pelo IntelliSense.

1.6.1.1. Compilando e executando o projeto

No Solution Explorer, clique com o botão direito do mouse sobre o nome do projeto e selecione **Build** ou **Rebuild**.



- **Build**: Compila somente os módulos que foram alterados;
- **Rebuild**: Recompila todos os módulos do projeto.

Para executar o código, podemos utilizar a opção **Start Debugging** ou **Start Without Debugging**, do menu **Debug**.



- **Start Debugging:** Permite executar o programa passo a passo;
- **Start Without Debugging** Permite executar o programa integralmente sem utilizar o debug.

Em seguida, no prompt, basta digitarmos o nome do programa seguido por **.exe**.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A linguagem C# é utilizada para a criação de vários tipos de aplicativos, como de banco de dados, cliente-servidor, aplicativos tradicionais do Windows, além de componentes distribuídos, XML Web Services, entre outros. Trata-se de uma linguagem orientada a objeto, com tipagem segura, cuja implementação em sistemas Microsoft é o Visual C#;
- A plataforma .NET oferece ferramentas para criação, distribuição e utilização de soluções de alta segurança e conectadas a serviços Web. Ela possibilita, ainda, a integração de diferentes sistemas de maneira eficiente;
- O Visual Studio IDE (Integrated Development Environment) é o ambiente de desenvolvimento utilizado para escrever programas em C#. Ele possui todas as ferramentas necessárias para desenvolver programas sólidos para o Windows e para a Web de forma rápida e eficiente;
- Para transformar um código digitado em um programa que pode ser executado, esse código deve ser compilado. Após a compilação do código, podemos executá-lo. Para isso, na janela do Visual Studio, podemos utilizar a opção **Start Debugging** ou **Start Without Debugging**, do menu **Debug**. No prompt, basta digitarmos o nome do programa seguido por **.exe**.

1

Introdução ao Visual Studio e C#

Teste seus conhecimentos

CarloS 77.3579963
SOUZA 007-93



IMPACTA
EDITORA

1. Qual é a melhor definição de classe de objeto?

- a) Uma entidade teórica, ou seja, que não existe fisicamente e não ocupa lugar na memória.
- b) Um componente que arrastamos da ToolBox para o formulário.
- c) Algo que está alocado na memória.
- d) Um conjunto de propriedades e métodos que estão alocados na memória.
- e) Um formulário herdeiro da classe Form.

2. Qual o elemento de uma classe que executa uma ação?

- a) Propriedade
- b) Evento
- c) Método
- d) Evento
- e) Herança

3. Qual o elemento de uma classe que armazena as características do objeto?

- a) Propriedade
- b) Evento
- c) Método
- d) Evento
- e) Herança

4. Qual o elemento de uma classe capaz de detectar uma ação externa sofrida pelo objeto?

- a) Propriedade
- b) Evento
- c) Método
- d) Evento
- e) Herança

5. Qual o nome da janela do Visual Studio que contém os componentes que colocamos sobre o formulário?

- a) Properties
- b) Solution Explorer
- c) Server Explorer
- d) ToolBox
- e) Windows Explorer

Formulários

2

- ✓ Criação de interface;
- ✓ Conceitos importantes;
- ✓ Controles de formulário;
- ✓ Resumo dos principais controles e suas propriedades;
- ✓ Adicionando menus;
- ✓ Adicionando barras de ferramentas;
- ✓ Caixas de diálogo padrão.



IMPACTA
EDITORA

2.1. Introdução

Os **Windows Forms** são formulários em branco utilizados em aplicações GUI (Graphic User Interface – Interface de Usuário Gráfica) que, em conjunto com os controles da ToolBox oferecida pelo Visual Studio, constituem uma maneira rápida e fácil de desenvolver aplicações profissionais. Os formulários e todas as classes relativas à interface Windows estão no namespace **System.Windows.Forms**.

Normalmente, os formulários são os elementos que promovem a interação com a interface de uma aplicação de Windows. Eles são chamados de janelas pelos usuários. Isso quer dizer que o que, o que é possível de ser visto pelo usuário recebe o nome de janela, enquanto o que visualizamos quando o programa está em desenvolvimento recebe o nome de formulário, porém, ambos os termos referem-se ao mesmo objeto.

Usamos formulários para que o usuário possa inserir ou obter informações a respeito do programa, seja por meio de texto, gráficos ou imagens. Além das configurações que ajustamos nos elementos colocados nos formulários, é permitido que eles próprios sofram ajustes.

2.2. Criando uma interface

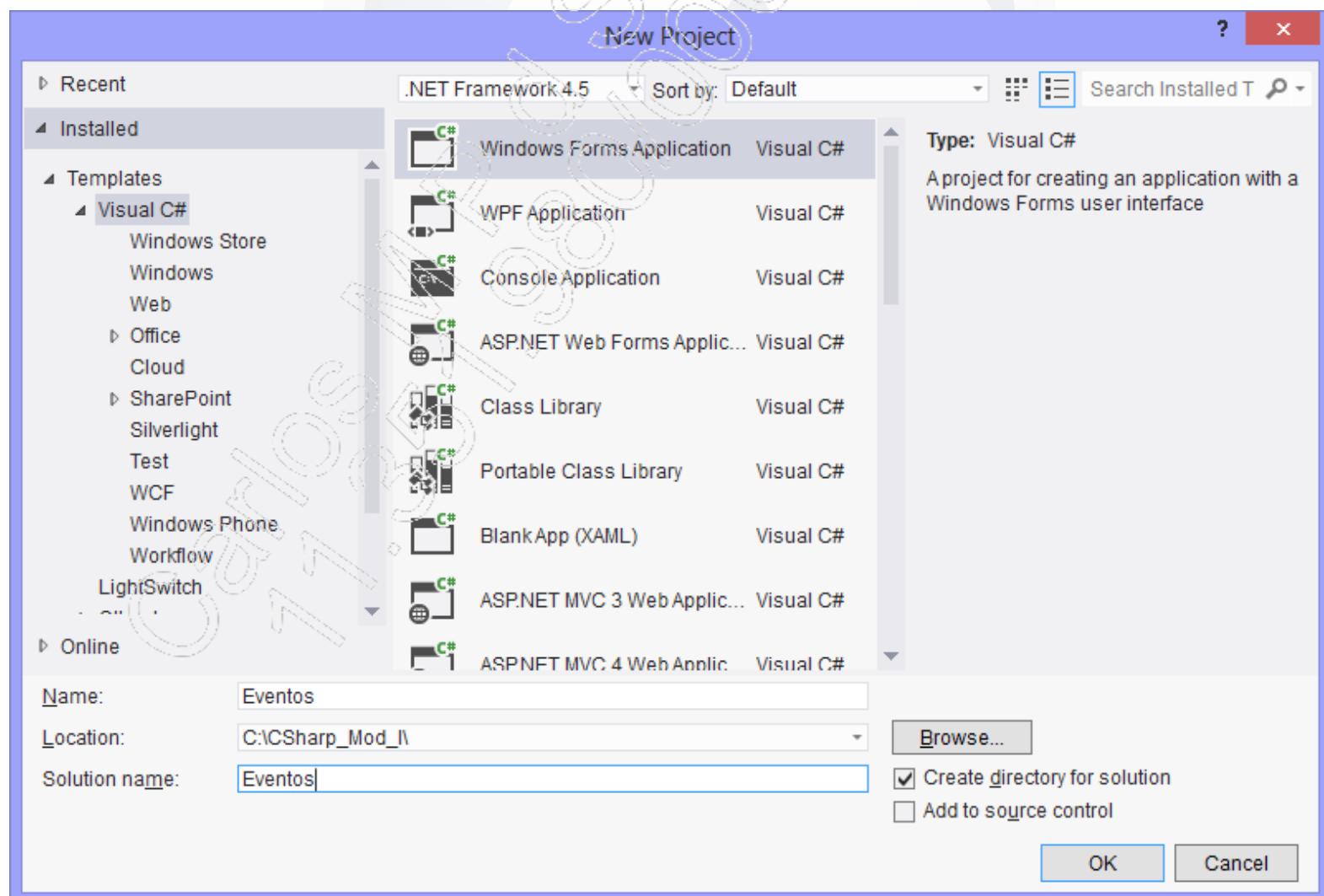
Um dos projetos mais comuns criados por meio do Visual Studio é a interface de usuário. Os procedimentos a seguir descrevem como utilizar a linguagem C# para criar uma interface de usuário Windows:

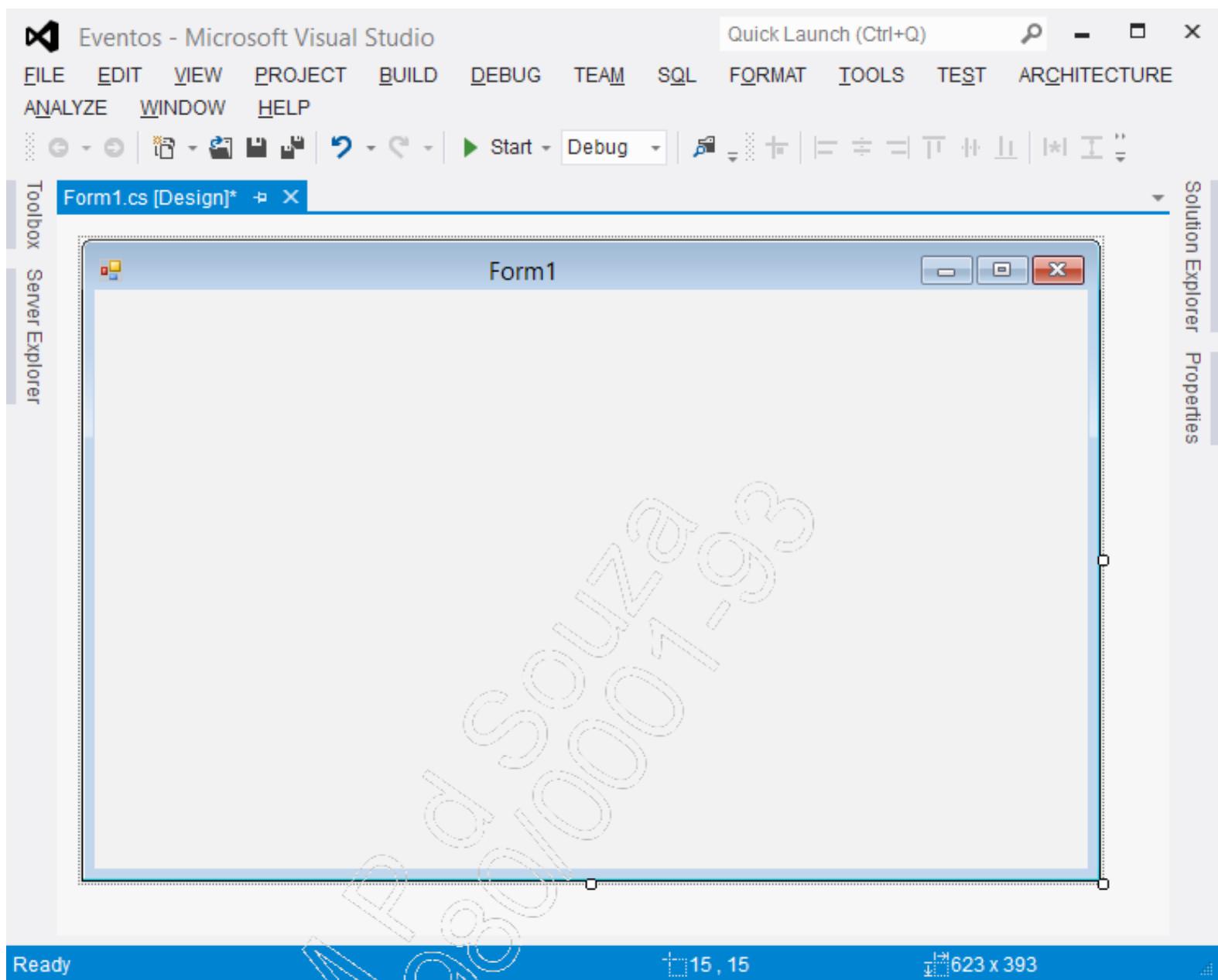
1. Clique em **File**;
2. Aponte para **New**;
3. Clique em **Project**;
4. Escolha **Visual C#** na lista de tipos de projetos;
5. Escolha **Windows Forms Application** na lista de templates;
6. Digite um nome para o projeto na caixa de texto **Name: Eventos**;

7. Escolha o local onde será salvo o projeto. Este passo é opcional. Se o local não for escolhido, o projeto é, por padrão, salvo na pasta **Visual Studio 2010**, em **Meus Documentos**;
8. Para o sistema criar uma pasta para a solução, mantenha a opção **Create directory for solution** marcada;
9. Clique em **OK**.

Um projeto sempre faz parte de uma solução, que pode possuir múltiplos projetos.

O resultado será o seguinte:





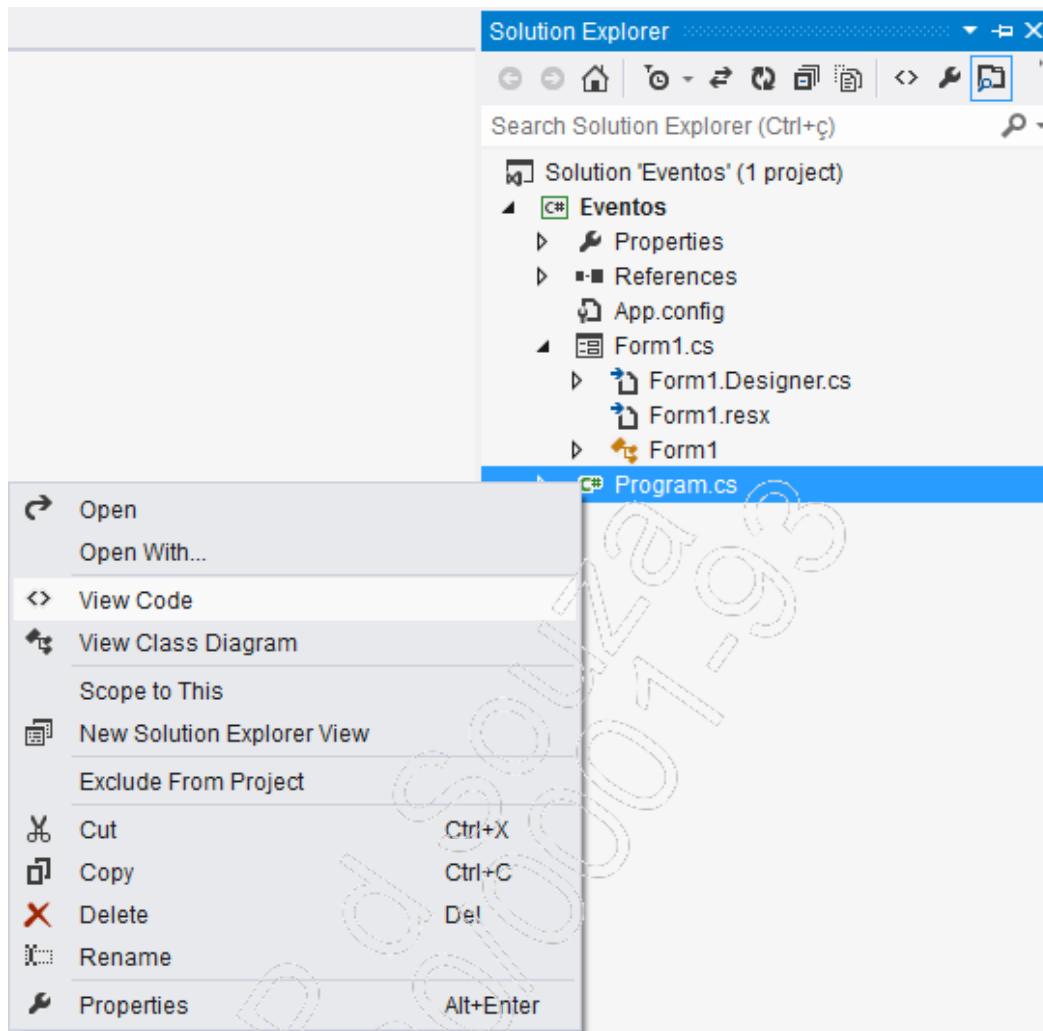
2.2.1. Formulário de inicialização

Por meio da instrução **Application.Run**, é possível exibir o formulário inicial de uma aplicação. Este, aliás, deve ser o único fim da instrução. Para a exibição de outros formulários, podemos utilizar o método **Show** herdado por quaisquer objetos **Windows Forms** para a visualização a partir do próprio código, normalmente em um manipulador de evento.

A **Application.Run** é tida como a instrução chave do método **Main**, o qual está localizado no arquivo **Program.cs**. Este pode ser visualizado a partir do **Solution Explorer**.

Para verificar o conteúdo de **Program.cs**, fazemos o seguinte:

1. No Solution Explorer, clique com o botão direito do mouse sobre o arquivo **Program.cs** e selecione **View Code**;



2. Verifique o conteúdo na janela **Code And Text Editor**.

A screenshot of the Visual Studio Code And Text Editor. The current file is 'Eventos.Program'. The code shown is the Main() method of the Program class, which is part of the 'Eventos' namespace. The code includes standard using statements for System, System.Collections.Generic, System.Linq, System.Threading.Tasks, and System.Windows.Forms. The Main() method contains a summary XML comment, the [STAThread] attribute, and the Application.Run() call.

A seguir, temos a sintaxe de **Application.Run**:

```
Application.Run(new Formulario());
```

Por meio da sintaxe anterior, uma nova instância de **Formulario** é criada e, em seguida, exibida. A instrução **Application.Run** será finalizada assim que o formulário for fechado. Consequentemente, o programa também será fechado, já que **Application.Run** é a instrução final de **Main**.

2.3. Conceitos importantes

Para um formulário ter utilidade, devemos acrescentar controles a ele. Antes de descrevê-los detalhadamente, porém, devemos abordar os conceitos de controle, objeto, propriedade, procedure de evento e método.

2.3.1. Controles

Para criar objetos em um formulário C#, utilizamos controles. Os controles são selecionados a partir da janela **ToolBox** e utilizados para desenhar, com o mouse, objetos em um formulário. A maioria desses controles é utilizada para gerar elementos de interface do usuário. Como exemplos desses elementos, temos os botões, as caixas de listagem e as imagens.

A maioria dos controles em .NET é derivada da classe **Control**, a qual define a funcionalidade básica de cada um dos controles. Encontramos **Control** no namespace **System.Windows.Forms**, o qual, por sua vez, está incluso nas diretrizes **using** em um dos arquivos que contém a classe **Form**.

2.3.2. Objetos

Podemos definir um objeto como um elemento de interface do usuário criado em um formulário C# com um controle da **ToolBox**. Em C#, o próprio formulário também é um objeto, e, com as configurações de propriedade, é possível movê-lo, redimensioná-lo e personalizá-lo.

Os objetos possuem uma característica conhecida como funcionalidade inherente, que permite que eles possam operar e responder a certas situações por iniciativa própria. Os objetos podem, ainda, ser programados utilizando procedures de evento personalizadas para situações variadas em um programa.

2.3.3. Propriedades

As propriedades são responsáveis por manipular o comportamento dos controles. Aqueles que são provenientes de **Control** herdam suas diversas propriedades. Para criar comportamentos personalizados, tais propriedades herdadas podem ser sobrepostas.

Também podemos definir uma propriedade como sendo um valor ou uma característica mantida por um objeto. Para citar um exemplo, um objeto (botão) possui uma propriedade **Text**, que tem a função de definir o texto que é exibido no botão.

Existem duas formas de configurar as propriedades em C#:

- Em tempo de projeto, por código, durante o desenvolvimento do programa, na janela **Properties**;
- Em tempo de execução, utilizando expressões no código do programa.

Em código, utiliza-se o formato adiante para configurar uma propriedade:

```
Objeto.Propriedade = Valor;
```

Os argumentos da sintaxe que acabamos de descrever são os seguintes:

- **Objeto**: Representa o nome do objeto que está sendo personalizado;
- **Propriedade**: Corresponde à característica que pretendemos alterar;
- **Valor**: Representa a nova configuração de propriedade.

O código a seguir exemplifica essa sintaxe:

```
textBox1.BackColor = Color.Yellow;
```

2.3.4. Métodos de evento

Um método de evento é um bloco de código que entra em ação quando é associado a um evento disparado pelo objeto.

Como exemplo de método de evento, podemos considerar que, ao clicarmos no objeto **button1**, o método de evento **button1_Click** será executado.

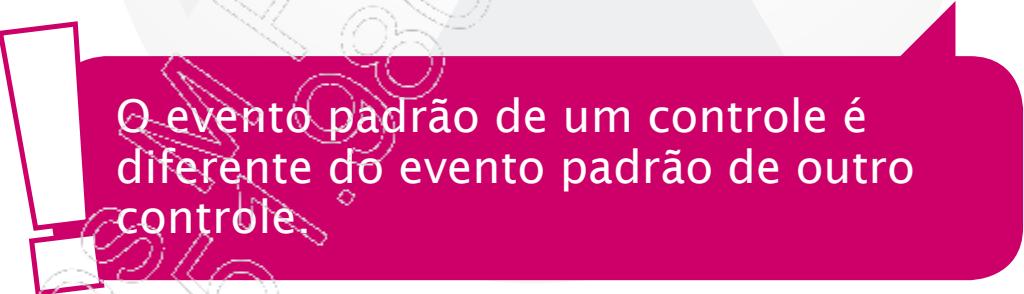
Geralmente, os métodos de evento avaliam e configuram propriedades. Tais métodos utilizam, ainda, outras expressões do programa.

A seguir, é apresentada uma tabela na qual consta a descrição dos eventos de controle mais comuns:

Evento	Descrição
Click	Ocorre quando o usuário clica sobre um controle ou, em alguns casos, quando a tecla ENTER é pressionada.
DoubleClick	Ocorre quando o usuário aplica um clique duplo sobre um controle.
DragDrop	Ocorre quando o usuário clica sobre um objeto e o arrasta e solta sobre um controle.
DragEnter	Ocorre quando o usuário arrasta um objeto até que ele entre nos limites de um controle.
DragLeave	Ocorre quando o usuário arrasta um objeto até que ele saia dos limites de um controle.
DragOver	Ocorre quando o usuário arrasta um objeto sobre um controle.
KeyDown	Ocorre quando o usuário pressiona uma tecla no momento em que o controle está com o foco. Ocorre antes de KeyPress e KeyUp .
KeyPress	Ocorre quando o usuário pressiona uma tecla no momento em que o controle está com o foco. Ocorre depois de KeyDown e antes de KeyUp . Enquanto KeyDown passa o código da tecla que foi pressionada, KeyPress passa o valor char de tal tecla. Somente teclas de texto disparam o evento KeyPress . São consideradas teclas de texto todas as teclas que têm imagem e mais o ENTER, o ESC e o BACKSPACE.
KeyUp	Ocorre quando o usuário solta uma tecla no momento em que o controle está com o foco. Ocorre após KeyDown e KeyPress .

Evento	Descrição
MouseDown	Ocorre quando o botão do mouse é pressionado no momento em que o cursor está sobre um controle. Diferente de Click , esse evento ocorre antes que o botão do mouse seja liberado.
MouseMove	Ocorre durante a movimentação do mouse sobre um controle.
MouseUp	Ocorre quando o botão do mouse é liberado no momento em que o cursor está sobre um controle.
Paint	Ocorre quando um controle é desenhado.
Validated	Disparado quando um controle que tem a propriedade CausesValidation configurada para true está prestes a receber o foco. É disparado depois que Validating é finalizado e indica que a validação foi completada.
Validating	Disparado quando um controle que tem a propriedade CausesValidation configurada para true está prestes a receber o foco. É importante lembrar que o controle que está perdendo o foco é o que será validado.

Para manipular um determinado evento, devemos aplicar um clique duplo sobre um controle. Essa ação faz com que sejamos conduzidos ao manipulador de evento padrão do controle no qual clicamos.



O evento padrão de um controle é diferente do evento padrão de outro controle

Se o evento padrão não for aquele com o qual desejamos trabalhar, podemos trocá-lo. Então, basta clicar no botão **Events** (⚡) presente na janela **Properties**.

Na lista **Events**, a seleção estará no evento padrão do controle, porém, podemos adicionar um manipulador para outro evento. Para tanto, basta aplicar um clique duplo sobre o nome do evento na lista **Events**, o que fará com que o código que subscreve o controle ao evento seja gerado, assim como a assinatura do método para manipular o evento.

Opcionalmente, podemos inserir um nome para o método que manipulará o evento, o que deve ser feito na própria lista **Events**, ao lado do evento desejado. Após inserir um nome, devemos pressionar a tecla ENTER, e o manipulador de evento será gerado.

Como uma segunda opção, podemos adicionar o código que deve subscrever o evento. No momento em que estivermos inserindo o código, o Visual Studio oferecerá a assinatura do método para o código, como se estivéssemos no **Forms Designer**.

Seja qual for a nossa escolha, dois procedimentos sempre serão obrigatórios:

- Subscrição do evento;
- Assinatura para o manipulador de evento.

Não podemos editar a assinatura do método do evento padrão com a finalidade de manipular outro evento, pois isso gera um erro; para que isso não ocorra, é necessário alterarmos também o código de subscrição do evento no **InitializeComponent()**.

Vejamos um exemplo:

```
this.sairButton.Click += new System.EventHandler(this.sairButton_Click);
```

2.3.5. Métodos

Nas situações em que se torna necessário executar uma ação ou um serviço para um objeto específico, podemos fazer uso das funcionalidades dos métodos.

Observemos, adiante, a sintaxe para a utilização de um método no código do programa:

```
Objeto.Metodo(Valor);
```

Ou

```
Classe.MetodoEstatico(Valor);
```

Os argumentos da sintaxe anterior são os seguintes:

- **Objeto:** Refere-se ao nome do objeto com que pretendemos trabalhar;
- **Classe:** Algumas classes definem métodos estáticos (marcados com **static**) que possibilitam que o método seja chamado sem uma instância, ou seja, diretamente pelo nome da classe;
- **Metodo:** Corresponde à ação que pretendemos executar;
- **Valor:** Refere-se a um argumento, opcional ou não, a ser utilizado pelo método.

O código a seguir exemplifica a sintaxe anterior:

```
textBox1.Text = Geral.RetirarAcentos(textBox2.Text);
```

2.4. Controles de formulário

Os controles podem ser definidos como ferramentas gráficas utilizadas para construir a interface de usuário de um programa C#.

Com relação à localização dos controles, eles estão na **Toolbox** do ambiente de desenvolvimento. Os segmentos presentes na **Toolbox**, normalmente chamados de guias, apresentam-se em ordem hierárquica. Quando clicamos sobre o símbolo + (sinal de adição) de uma das guias, ela é expandida e exibe diversos controles que podemos escolher. As guias podem ser expandidas todas ao mesmo tempo, uma vez que podemos rolar a lista de controles da **Toolbox** dentro da própria janela para visualizar todos os itens. Já o símbolo - (sinal de subtração) faz com que os controles da guia sejam ocultados.

Os itens que possuem ícones são aqueles que podemos adicionar ao projeto. Sempre que mudamos o foco para o editor ou designer ou passamos para outro projeto, a **Toolbox** passa a exibir os últimos controles usados naquele local.

Com os controles da **Toolbox**, criamos objetos em formulários por meio da simples sucessão de cliques e movimentos de arrastar e soltar.

A janela **ToolBox** é dividida em categorias para facilitar a utilização das ferramentas disponíveis. As principais categorias são as seguintes:

- **All Windows Forms**: Contém todos os controles para Windows, organizados em ordem alfabética;
- **Common Controls**: Contém os controles mais comuns para criar interfaces de usuários;
- **Containers**: Contém controles que servem para agrupar outros controles internamente, como painéis e grupos de opções;
- **Menus & Toolbars**: Contém controles para a criação de menus e barras de ferramentas;
- **Data**: Contém controles para exibir e vincular fontes de dados;
- **Components**: Contém controles para diversas funcionalidades, como o Timer, que dispara um evento em intervalos regulares, e o **ImageList**, que agrupa imagens para serem usadas em outros controles;
- **Printing**: Contém controles para impressão e visualização de documentos a serem impressos;
- **Dialogs**: Contém controles para criar caixas de diálogo comuns em aplicações Windows, como aquelas para escolha de cores e de fontes, ou aquelas para abrir ou salvar arquivos;
- **Reporting**: Contém controles para visualizar e criar relatórios;
- **General**: Esta divisão da ToolBox encontra-se, por padrão, vazia. É possível arrastar e soltar textos para serem reaproveitados.

Os controles mais comuns da ToolBox serão abordados adiante.

2.5. Resumo dos principais controles e suas propriedades

Neste tópico, vamos conhecer os principais controles de formulário e suas propriedades.

2.5.1. Label e LinkLabel

O controle gráfico **Label** é responsável por exibir um texto que não pode ser editado pelo usuário. No entanto, podemos atribuir para esse tipo de texto um código que faça com que ele seja trocado por outro, de acordo com os eventos que ocorrerem em tempo de execução. É muito comum usar esse tipo de controle nos casos em que se deseja exibir uma mensagem referente ao status do processamento de alguma ação. **Label** é bastante empregado, ainda, para identificar controles que não possuem uma propriedade com essa finalidade, como o **TextBox**.

Label é, possivelmente, o controle mais utilizado, visto que quase todas as caixas de diálogo apresentam um texto.

Além do controle **Label**, temos o **LinkLabel**. A diferença entre eles é que o primeiro apresenta um texto simples, enquanto o segundo apresenta um texto que é também um link para a Internet, ou hyperlink.

Quando trabalhamos com o controle **Label**, a atribuição de códigos de manipulação de evento não se faz necessária na maior parte das vezes, embora ele os possa receber como qualquer outro tipo de controle. Por outro lado, o controle **LinkLabel** exige o código que direcionará o usuário ao conteúdo sugerido pelo link.

2.5.1.1. Propriedades dos controles Label e LinkLabel

A seguir, temos uma tabela em que constam as principais propriedades dos controles **Label** e **LinkLabel**:

Propriedade	Descrição
AutoSize	Define se o objeto ajusta automaticamente seu tamanho.
BorderStyle	Define o estilo da borda que aparece em torno de Label . Por padrão, Label não exibe borda.
Dock	Define se Label fica fixo nos cantos ou no centro da tela.
FlatStyle	Define como o controle deve aparecer. Se for configurada para Popup , o controle permanecerá plano até que o cursor do mouse seja posicionado sobre ele, o que fará com que o controle passasse a ser exibido em alto-relevo.
Font	Define a fonte utilizada para mostrar o texto do objeto.
Image	Define uma imagem a ser exibida em Label , a qual pode ser um bitmap ou um ícone, entre outros.
ImageAlign	Define o posicionamento da imagem em Label .
LinkArea	Aplicável somente a LinkLabel , define a parte do texto que deve aparecer como um link.
LinkColor	Aplicável somente a LinkLabel , define a cor que o link deve ter.
LinksVisited	Aplicável somente a LinkLabel , exibe o texto do link em uma outra cor caso ele já tenha sido clicado.
Name	Define o nome do objeto.
Text	Define o texto a ser exibido.
 TextAlign	Define em que parte do controle o texto deve ser exibido.
VisitedLinkColor	Aplicável somente a LinkLabel , define a cor que o link deve exibir após ter sido clicado.

2.5.2. TextBox e RichTextBox

O tipo de informação mais comum que se insere no controle **TextBox** é o texto propriamente dito, porém, é possível inserir qualquer tipo de caractere e podemos, inclusive, restringir a entrada apenas a números, por exemplo.

Além do **TextBox**, temos disponível também o **RichTextBox**, ambos derivados da classe base **TextBoxBase**, a qual, por sua vez, é derivada de **Control**. A diferença entre esses dois controles é que o primeiro não aceita formatação caractere a caractere enquanto que o segundo pode ter tal formatação.

2.5.2.1. Propriedades dos controles TextBox e RichTextBox

A seguir, temos uma tabela em que constam as principais propriedades e métodos dos controles **TextBox** e **RichTextBox**:

Propriedade	Descrição
CausesValidation	Quando configurada para true , os eventos Validating e Validated são disparados no momento em que o controle estiver para receber o foco. Esses eventos validam os dados do controle que estiver perdendo o foco.
CharacterCasing	Define se o texto deve ser mantido em letras maiúsculas (Upper), em letras minúsculas (Lower) ou da forma como o usuário inserir (Normal).
Clear()	Esse método tem como finalidade limpar o texto.
MaxLength	Define a extensão máxima do texto a ser inserido no TextBox . Esse valor é definido em caracteres. Se for configurado como zero, o limite será estabelecido de acordo com a memória disponível.
Multiline	Define se TextBox deve apresentar mais de uma linha de texto. Se configurarmos essa propriedade como true , é recomendável também configurar como true a propriedade WordWrap .
Name	Define qual deve ser o nome do objeto.
PasswordChar	Define se os caracteres reais de uma senha devem ser substituídos por algum outro caractere (como o asterisco) quando exibidos na linha de texto. Essa propriedade não funciona quando Multiline está ativada.
ReadOnly	Define se o texto deve ser para somente leitura, evitando, assim, que seja editado.
ScrollBars	Define se uma caixa de texto que possua mais de uma linha deve apresentar barra de rolagem.
Text	Define o texto visível do objeto.
WordWrap	Define se a quebra de linha deve ser automática nas caixas de texto que possuem mais de uma linha.

2.5.2.2. Eventos do controle TextBox

Os eventos do controle **TextBox** são usados para indicar para o usuário se a quantidade e o tipo de caracteres inseridos em uma caixa de texto são válidos ou não, evitando, assim, que informações incorretas sejam digitadas.

A seguir, temos uma tabela com os principais eventos que **TextBox** disponibiliza:

Evento	Descrição
Enter Leave Validating Validated	Esses quatro eventos, conhecidos como eventos de foco, são disparados sempre que um controle perde o foco. Porém, Validating e Validated são disparados apenas quando o controle que estiver recebendo o foco tiver a propriedade CausesValidation configurada como true .
KeyDown KeyPress KeyUp	Esses três eventos, conhecidos como eventos de teclado, conferem-nos o poder de acompanhar e alterar as informações inseridas no controle. Os eventos KeyDown e KeyUp são responsáveis por receber o código referente à tecla pressionada, o que nos permite saber se teclas especiais como SHIFT ou CTRL foram pressionadas pelo usuário. Já o KeyPress é responsável por receber o caractere referente à tecla pressionada. A partir daí, concluímos que A não possui o mesmo valor que A. Isso será bastante útil nos casos em que desejarmos restringir o tipo de caractere que pode ser inserido em uma caixa de texto.
TextChanged	Ocorre quando há qualquer tipo de alteração no texto presente na caixa de texto.

2.5.3. Button

Este controle é responsável por induzir a aplicação a executar uma tarefa predefinida e está presente em quase todas as caixas de diálogo do Windows. O controle **Button** pode executar um dos seguintes tipos de ações:

- Fechar uma caixa de diálogo;
- Abrir uma caixa de diálogo ou aplicação;
- Executar uma ação usando a informação inserida na caixa de diálogo.

O .NET Framework disponibiliza a classe **System.Windows.Forms.ButtonBase**, derivada de **Control**, que é capaz de implementar a funcionalidade básica de controles **Button**. Assim, podemos criar controles **Button** personalizados a partir da derivação dessa classe.

Para usar um controle do tipo **Button**, devemos simplesmente adicioná-lo ao formulário, aplicar um clique duplo sobre ele e adicionar o código ao evento **Click**.

2.5.3.1. Propriedades do controle Button

A seguir, temos uma tabela com as principais propriedades do controle **Button**:

Propriedade	Descrição
Anchor	Define a posição do controle em relação às bordas do formulário.
Enabled	Essa propriedade define se o botão deve permanecer habilitado ou desabilitado.
FlatStyle	Essa propriedade define o estilo do botão. Se for configurada para Popup , o botão permanecerá plano até que o cursor do mouse seja posicionado sobre ele, o que faz com que passe a ser exibido em alto-relevo, ou 3D.
Image	Define uma imagem para ser exibida no botão. Essa imagem pode ser um bitmap ou um ícone, por exemplo.
ImageAlign	Define o posicionamento da imagem no botão.
Name	Define o nome do controle.
Text	Define o texto visível do controle.

2.5.3.2. Evento do controle Button

Normalmente, o evento que usamos com **Button** é o **Click**, que ocorre sempre que o usuário posiciona o cursor do mouse sobre o controle e pressiona e solta o botão esquerdo do mouse. Se o cursor for movido para outro local que não seja sobre o controle, antes que o usuário tenha soltado o botão do mouse, então o evento **Click** não será disparado. Por outro lado, mesmo que não haja o clique do mouse, se o controle estiver com o foco, basta pressionar ENTER e o evento será disparado. Para os formulários que possuam botões, o uso do evento **Click** será indispensável.

2.5.4. RadioButton

Responsável por fazer com que o usuário escolha apenas um item a partir de uma lista de possibilidades, o controle **RadioButton** é exibido como um rótulo que traz, normalmente à sua esquerda, um pequeno círculo que podemos ou não selecionar.

Para criar um grupo com mais de um controle do tipo **RadioButton**, devemos usar um contêiner, como o controle **GroupBox**, por exemplo. Quando utilizarmos um contêiner em um formulário, apenas um dos controles **RadioButton** que inserirmos nele poderá ser selecionado. Assim, se tivermos diversos grupos em um formulário, poderemos selecionar uma opção de cada um deles. No entanto, se não usarmos o contêiner, apenas uma das opções de todo formulário poderá ser selecionada.

2.5.4.1. Propriedades do controle RadioButton

Na tabela a seguir, temos a descrição das principais propriedades do controle **RadioButton**:

Propriedade	Descrição
Appearance	Define a aparência do controle, que pode ser ou um rótulo com um pequeno círculo ao lado esquerdo, direito ou no centro, ou como um botão padrão, o qual apresenta aparência de pressionado quando o selecionamos.
AutoCheck	Essa propriedade pode ser definida como true , para que um ponto preto apareça quando o usuário clicar no controle, ou como false , o que exige que o botão seja assinalado manualmente no código a partir do manipulador de evento Click .
CheckAlign	Essa propriedade, configurada como ContentAlignment.MiddleLeft , define o alinhamento da caixa de seleção do controle RadioButton .
Checked	Essa propriedade define o status do controle.
Name	Essa propriedade define o nome do objeto.
Text	Essa propriedade define o texto visível do objeto.

2.5.4.2. Eventos do controle RadioButton

Normalmente, utilizamos apenas um evento com o controle **RadioButton**. Porém, é possível subscrever outros. A tabela a seguir descreve os dois principais eventos utilizados com **RadioButton**:

Evento	Descrição
CheckedChanged	Disparado quando a seleção do RadioButton for alterada.
Click	Disparado quando o controle RadioButton é clicado.

 Não devemos confundir o evento **Click** e o evento **CheckedChange**. Ao clicarmos no controle **RadioButton** repetidas vezes, a propriedade **Checked** não será alterada mais de uma vez.

2.5.5. CheckBox

Responsável por exibir uma lista de opções que podem ser selecionadas simultaneamente, o controle **CheckBox** é exibido como um rótulo que traz, normalmente à sua esquerda, uma pequena caixa que podemos ou não selecionar.

2.5.5.1. Propriedades do controle CheckBox

Na tabela a seguir, estão descritas as principais propriedades do controle **CheckBox**:

Propriedade	Descrição
Checked	Essa propriedade define se o controle está selecionado ou não.
CheckedAlign	Essa propriedade define o alinhamento da caixa de seleção em relação ao texto.
CheckedState	Define qual é o estado do CheckBox , que pode ser Checked (selecionado), Unchecked (não selecionado) e Indeterminate (um dos valores atuais da seleção não é válido ou não faz sentido em um determinado contexto).
Name	Essa propriedade define o nome do objeto.
Text	Essa propriedade define o texto visível do objeto.
ThreeState	Quando configurada como false , essa propriedade impedirá que o usuário defina CheckState como Indeterminate . A alteração da propriedade CheckState para Indeterminate poderá ser feita somente por meio de código.

2.5.5.2. Eventos do controle CheckBox

Normalmente, utilizamos apenas um ou dois eventos com o controle **CheckBox**. A tabela a seguir descreve dois eventos utilizados com **CheckBox**:

Evento	Descrição
CheckedChanged	É disparado quando a propriedade Checked da caixa de seleção é alterada. Se tivermos um CheckBox em que a propriedade ThreeState esteja definida como true , o clique na caixa de seleção não refletirá alterações na propriedade Checked (isso ocorre nas situações em que a caixa de seleção tem seu status alterado de Checked para Indeterminate).
CheckStateChanged	É disparado quando a propriedade CheckedState é alterada. Também será disparado quando a propriedade Checked for alterada, uma vez que Checked e Unchecked são valores da propriedade CheckedState . Será disparado, ainda, quando o estado passar de Checked para Indeterminate .

2.5.6. ListBox

Responsável por exibir uma lista de opções que o usuário pode escolher. O controle **ListBox** é exibido como uma lista de strings. É indicado para os casos em que há um grande número de opções possíveis, permitindo ao usuário selecionar uma ou mais delas.

A classe **ListBox** é derivada da classe **ListControl**, a qual fornece para **ListBox** as funcionalidades básicas dos controles do tipo lista.

2.5.6.1. Propriedades do controle ListBox

Na tabela a seguir, estão descritas as principais propriedades do controle **ListBox**:

Propriedade	Descrição
CheckOnClick	Aplicável somente a CheckedListBox , quando essa propriedade está definida como true , faz com que o estado de um item seja alterado assim que o usuário clica sobre ele.
ThreeDCheckBoxes	Aplicável somente a CheckedListBox , define se CheckBoxes devem ter aparência em alto-relevo ou normal.
ColumnWidth	Define a largura das colunas, quando a lista possuir diversas delas.
Items	Trata-se de uma coleção em que estão todos os itens da lista.
Name	Define o nome do objeto.
MultiColumns	Define se os valores de uma lista devem ser exibidos em mais de uma coluna.
SelectionMode	Essa propriedade define o tipo do modo de seleção de ListBox , que pode ser None (nenhum item pode ser selecionado), One (apenas um item pode ser selecionado), MultiSimple (mais de um item pode ser selecionado, e um item só perde a seleção quando for clicado de novo) e MultiExtended (mais de um item pode ser selecionado, e um item perde a seleção quando clicamos em outro sem manter pressionadas as teclas CTRL + SHIFT).
Sorted	Essa propriedade, se configurada como true , faz os itens contidos na lista serem disponibilizados em ordem alfabética.

2.5.6.2. Métodos do controle ListBox

Na tabela a seguir, estão descritos os principais métodos do controle **ListBox**, os quais estão contidos, em sua maioria, nas classes **ListBox** e **CheckedListBox**:

Método	Descrição
ClearSelected()	Esse método limpa as seleções da ListBox .
FindString()	Com esse método, podemos encontrar a primeira string da ListBox que iniciar com uma string que nós informarmos, como a letra T, por exemplo.
FindStringExact()	Com esse método, podemos encontrar a string que coincide com a string que informarmos. Diferente de FindString , com FindStringExact é necessário especificar a string inteira, e não só a primeira letra.
GetSelected()	Esse método retorna um valor que indica se um item está selecionado.
SetSelected()	Esse método configura ou limpa a seleção de um item.
ToString()	Esse método retorna o item selecionado atualmente.
GetItemChecked()	Disponível apenas para CheckedListBox , esse método retorna um valor que indica se um item está marcado.
GetItemCheckState()	Disponível apenas para CheckedListBox , esse método retorna um valor que indica o estado de um item.
SetItemChecked()	Disponível apenas para CheckedListBox , esse método configura o item que especificarmos para o estado Checked .
SetItemCheckState()	Disponível apenas para CheckedListBox , esse método configura o estado de um item.

2.5.7. ComboBox

Esse controle usa, em conjunto, um campo de texto e uma lista. Tendo disponíveis esses dois elementos, o usuário pode ou digitar um texto, ou selecionar um item da lista. Por padrão, o **ComboBox** aparece como um campo de texto com uma lista drop-down oculta.

2.5.7.1. Propriedades do controle ComboBox

Na tabela seguinte, temos a descrição das principais propriedades de **ComboBox**:

Propriedade	Descrição
DropDownStyle	Essa propriedade define o estilo do controle. Podemos configurá-lo de modo que seja exibida ou uma lista drop-down simples, o que mantém a lista sempre visível; ou uma lista drop-down em que o texto não pode ser editado pelo usuário, e as opções somente podem ser visualizadas após o clique sobre uma seta; ou uma lista drop-down padrão, a qual permite edição no texto e exibe as opções somente após o clique em uma seta.
Items	Trata-se de uma coleção de objetos na lista.

2.5.8. DateTimePicker

Esse controle permite a seleção de uma data a partir de um calendário que é aberto quando o usuário clica sobre uma seta. Vamos conhecer, a seguir, as propriedades desse controle.

2.5.8.1. Propriedades do controle DateTimePicker

Na tabela a seguir, temos a descrição das principais propriedades de **DateTimePicker**:

Propriedade	Descrição
Format	Essa propriedade define o formato da data a ser mostrada.
Name	Essa propriedade define o nome do objeto.
Value	Essa propriedade contém a data selecionada.

2.5.9. TabControl

Este controle permite criar guias, as quais têm como objetivo organizar um formulário em diferentes partes, normalmente seguindo uma estrutura lógica.

O controle **TabControl** possui **TabPage**s, que são as guias responsáveis por agrupar os diferentes controles. Após adicionar a quantidade de guias com que se deseja trabalhar, basta arrastar os controles desejados para cada uma delas.

2.5.9.1. Propriedades do controle TabControl

Na tabela a seguir, temos a descrição das principais propriedades do controle **TabControl**:

Propriedade	Descrição
Alignment	Define a localização das guias dentro de TabControl . Por padrão, está configurada para exibir as guias no topo.
Appearance	Define a aparência das guias, que pode ser normal ou em alto-relevo.
Hot-Track	Essa propriedade, quando configurada como true , faz com que a aparência das guias mude quando o ponteiro do mouse for posicionado sobre elas.
Multiline	Essa propriedade, quando configurada como true , permite que seja exibida mais de uma linha de guias.
Name	Essa propriedade define o nome do objeto.
SelectedTab	Disponível somente para as instâncias atuais de TabPage , essa propriedade configura ou retorna a guia selecionada.
TabPage s	Essa propriedade representa a coleção de objetos TabPage presentes no controle, ou seja, é o conjunto de guias disponíveis para o usuário, e pode ser usada para adicioná-las ou removê-las.

2.5.10. Timer

Com a utilização do controle **Timer**, podemos executar um grupo de expressões para um período de tempo específico ou em intervalos específicos. Além disso, ele pode ser utilizado para realizar a contagem regressiva a partir de um tempo presente, repetir uma ação em intervalos fixos ou causar um atraso em um programa.

2.5.10.1. Propriedades do controle Timer

Na tabela a seguir, temos a descrição das principais propriedades de **Timer**:

Propriedade	Descrição
Enabled	Essa propriedade habilita o funcionamento do timer.
Interval	Essa propriedade define o intervalo, em milésimos de segundos, em que ocorrerá um evento.

2.6. Adicionando menus

A construção de menus e de barras de ferramentas com aparência profissional é uma tarefa simples e rápida de ser feita, por meio dos controles **MenuStrip** e **ToolStrip**.

Juntos, os controles **MenuStrip** e **ToolStrip** formam uma família de controles que trabalham em harmonia, sendo que **MenuStrip** deriva diretamente de **ToolStrip**.

2.6.1. MenuStrip

Os menus podem ser acrescentados nos programas com a utilização de um controle chamado **MenuStrip**.

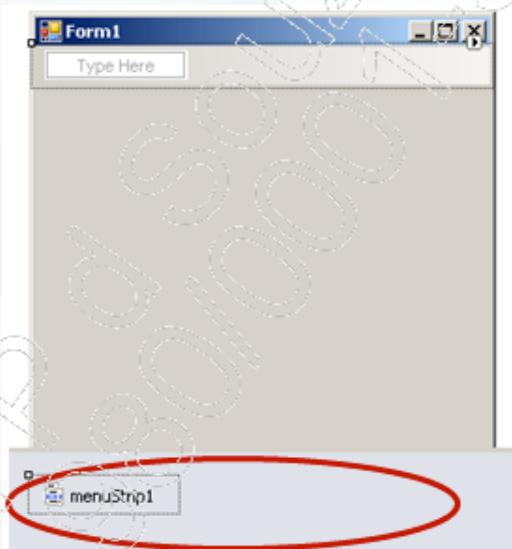
O controle **MenuStrip** permite não apenas adicionar novos menus, mas também modificar, reordenar e excluir aqueles já existentes. Junto a essas possibilidades, há, ainda, a capacidade de implementar os menus com efeitos – por exemplo, teclas de atalho, chaves de acesso e marcas de verificação –, bem como definir configurações de menu padrão de forma automática.

Por meio das configurações de propriedade disponíveis na janela **Properties**, torna-se possível personalizar as configurações do menu que é adicionado ao programa.

Sobre a visualização dos menus, devemos ter em mente que apenas parte deles e dos comandos torna-se visível com a utilização do **MenuStrip**.

O objeto **MenuStrip** aparece abaixo do formulário. Neste ponto, vale ressaltar que os objetos invisíveis disponíveis no Visual Studio são exibidos no IDE, em um painel separado denominado **Component Tray**. Tais objetos podem ser selecionados ou removidos, bem como ter suas propriedades configuradas.

A imagem a seguir mostra o painel **Component Tray** abaixo do formulário **Form1**:



Outro aspecto importante a ser considerado na criação de menus é que uma representação visual do menu criado é exibida pelo Visual Studio na parte superior do formulário, e isso surge em acréscimo ao **MenuStrip** presente no **Component Tray**.

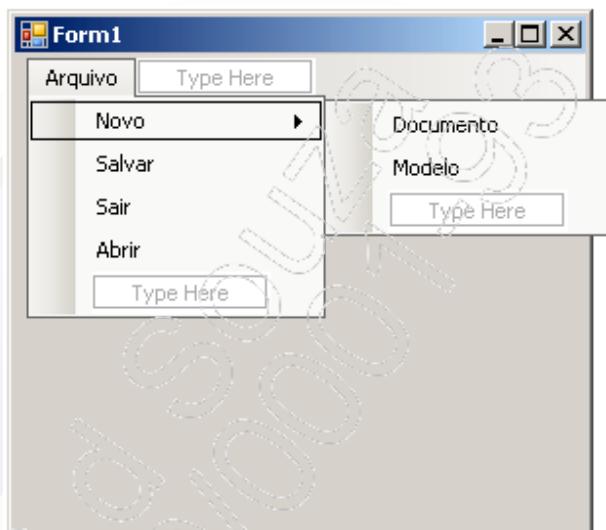
Ao clicarmos na guia onde está escrito **Type Here**, podemos definir um título para o menu. Assim que especificamos um título inicial, podemos definir títulos para possíveis submenus, bem como outros nomes para novos menus. Isto é feito por meio das setas direcionais e da digitação dos novos nomes.

Um aspecto bastante vantajoso desse ambiente destinado à construção de menus, é que o desenvolvedor pode retornar posteriormente para realizar alterações nos menus já criados e, até mesmo, acrescentar novas estruturas de menus e/ou submenus. O objeto **MenuStrip** é totalmente personalizável.

2.6.2. ToolStrip

Enquanto o **MenuStrip** representa o contêiner da estrutura do menu de um formulário, os objetos **ToolStripMenuItem** representam os comandos contidos na estrutura do menu ou, ainda, um menu pai para outros itens.

Na imagem seguinte, está ilustrada a construção de um menu. O **MenuStrip** chamado **Arquivo** contém os controles **ToolStrip** denominados **Novo**, **Salvar**, **Sair** e **Abrir**.



2.6.2.1. Propriedades do controle **MenuStrip**

As duas principais propriedades de **MenuStrip** estão descritas na tabela a seguir:

Propriedade	Descrição
Name	Define o nome do objeto.
Items	Representa a coleção de itens do menu.

2.6.2.2. Propriedades do controle **ToolStrip**

As duas principais propriedades de **ToolStrip** estão descritas na tabela a seguir:

Propriedade	Descrição
Name	Define o nome do objeto.
Items	Representa a coleção de botões na barra de ferramentas.

2.6.3. Configurando as teclas de acesso para os comandos do menu

O teclado é o dispositivo utilizado para acessar e executar comandos de menu na maioria das aplicações. É possível, por exemplo, acessar o menu **File** do Visual Studio combinando as teclas ALT + F. Para abrir um projeto depois que o menu é acessado, basta pressionar a tecla P.

A possibilidade de combinar teclas com ALT ou de pressionar teclas separadamente para executar um comando do menu gera uma característica especial: essas teclas, conhecidas como teclas de acesso, encontram-se sublinhadas no menu, o que permite identificá-las facilmente.

O processo para criar uma tecla de acesso para um menu é simples. Basta ativar o **Menu Designer** e digitar & antes da letra desejada presente no nome do menu, como vemos nas imagens a seguir. O suporte à tecla de acesso é fornecido automaticamente ao programa no momento em que este é carregado.



Ressaltamos que, em alguns sistemas operacionais, por padrão, a marca sublinhada só é exibida no programa assim que a tecla ALT é pressionada pela primeira vez. No entanto, tal comportamento pode ser desabilitado por meio das propriedades de vídeo do Windows.

2.6.4. Convenções para criar menus

Existem algumas convenções que costumam ser empregadas durante a criação de menus. Uma delas refere-se ao fato de as aplicações Windows geralmente terem os títulos e os comandos de menu especificados com a letra inicial em caixa alta, ou seja, maiúscula.

Também há convenções comumente empregadas para definir a posição de cada menu na barra. Por exemplo, os menus de nome **Arquivo** e **Editar** geralmente são dispostos na parte inicial da barra de menus, ao passo que o menu chamado **Ajuda** costuma estar localizado na parte final da barra. No entanto, o que realmente importa na disposição dos menus e dos comandos em uma aplicação é a consistência e clareza que possuem, pois eles devem ser assimilados facilmente pelos usuários.

Há algumas medidas que podem ser adotadas para a criação adequada de menus, conforme apresentado adiante:

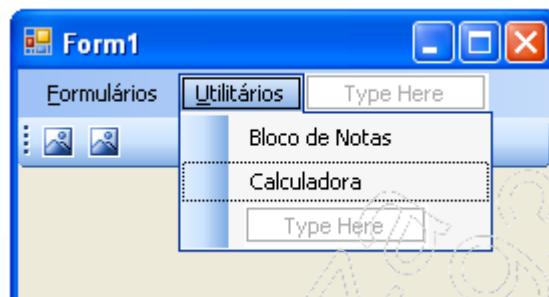
- Os títulos devem ser objetivos, formados por não mais que duas palavras;
- É importante definir uma chave de acesso para cada um dos menus, sendo a letra inicial deles a mais indicada para ser a chave, exceto em alguns casos, quando outras letras costumam ser utilizadas como convenção (por exemplo, o **b** para **Tabela**);
- Apenas uma chave de acesso deve ser definida para os itens de menus dispostos em um mesmo nível;
- Em se tratando de menus que podem ser ativados ou desativados, é importante definir uma marca de verificação à esquerda do item, para que ela seja exibida quando o item for ativado. A ativação da marca de verificação pode ser feita configurando-se como **True** a propriedade **Checked** do comando de menu, disposta na janela **Properties**;
- Quando, ao ser clicado, um item de menu redireciona para uma caixa de diálogo, devemos acrescentar um sinal de reticências (...) ao final do item para que os usuários saibam que ele conduzirá a uma nova janela.

2.6.5. Executando as opções do menu

Depois de criar os menus e comandos da aplicação, é necessário escrever procedimentos de evento para fazê-los funcionar. Isso porque eles se tornam novos objetos no programa depois que são configurados por meio do **MenuStrip**.

Os procedimentos de evento geralmente são escritos com instruções que, além de alterarem uma ou mais propriedades de menu, exibem ou processam dados no formulário referente à interface de usuário. Para criar um método que seja executado quando o usuário clicar em um item de menu ou escolher uma tecla de atalho, basta aplicar um clique duplo sobre o menu.

Podemos visualizar um exemplo a seguir:

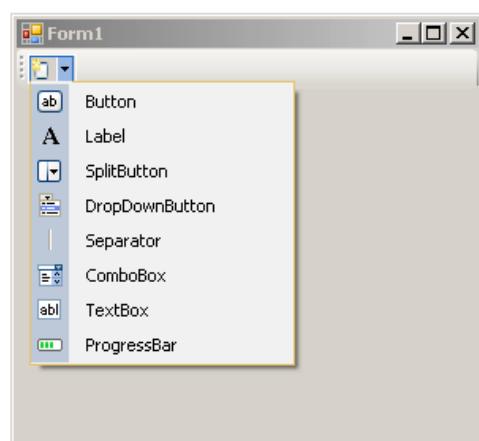


```
private void calculadoraToolStripMenuItem_Click(object sender, EventArgs e)
{
    System.Diagnostics.Process.Start("calc.exe");
}
```

2.7. Adicionando barras de ferramentas

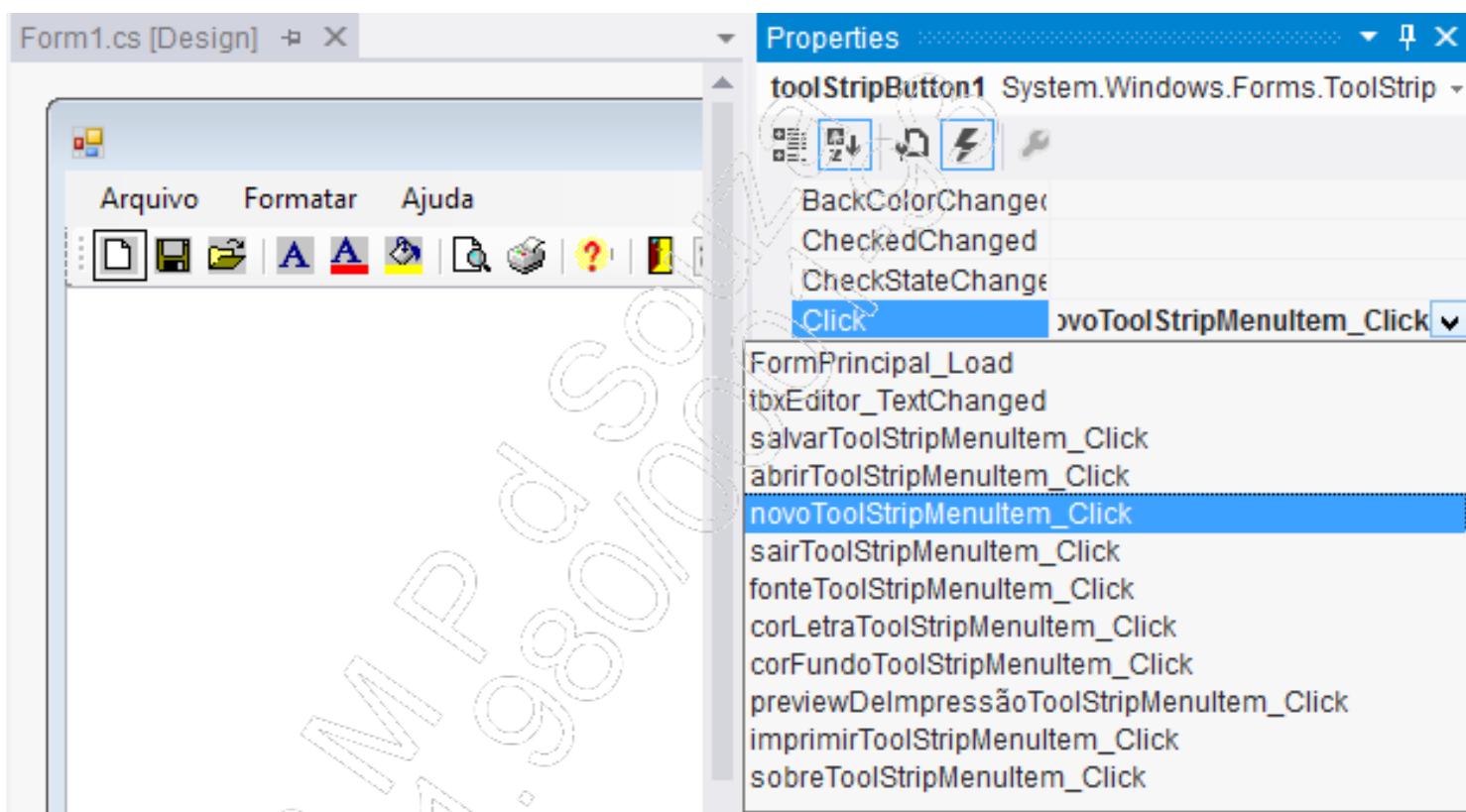
Podemos adicionar barras de ferramentas a uma aplicação por meio do controle **ToolStrip**.

Além dos botões, podemos adicionar os controles **Label**, **SplitButton**, **DropDownButton**, **Separator**, **ComboBox**, **TextBox** e **ProgressBar** às barras de ferramentas, como mostrado na imagem seguinte:



Depois de criar a barra de ferramentas da aplicação, é necessário escrever procedures de evento para fazê-la funcionar, assim como ocorre nos comandos de menu. Tais procedimentos de evento devem ser definidos para cada um dos botões da barra de ferramentas.

Para evitar a duplicação de código, é interessante manter os comandos do programa apenas dentro dos menus e usar a barra de ferramentas como atalho. Sendo assim, vá até o evento **Click** do botão na barra de ferramentas, mas em vez de executar um duplo-clique, clique na setinha e selecione o método já criado para o menu.



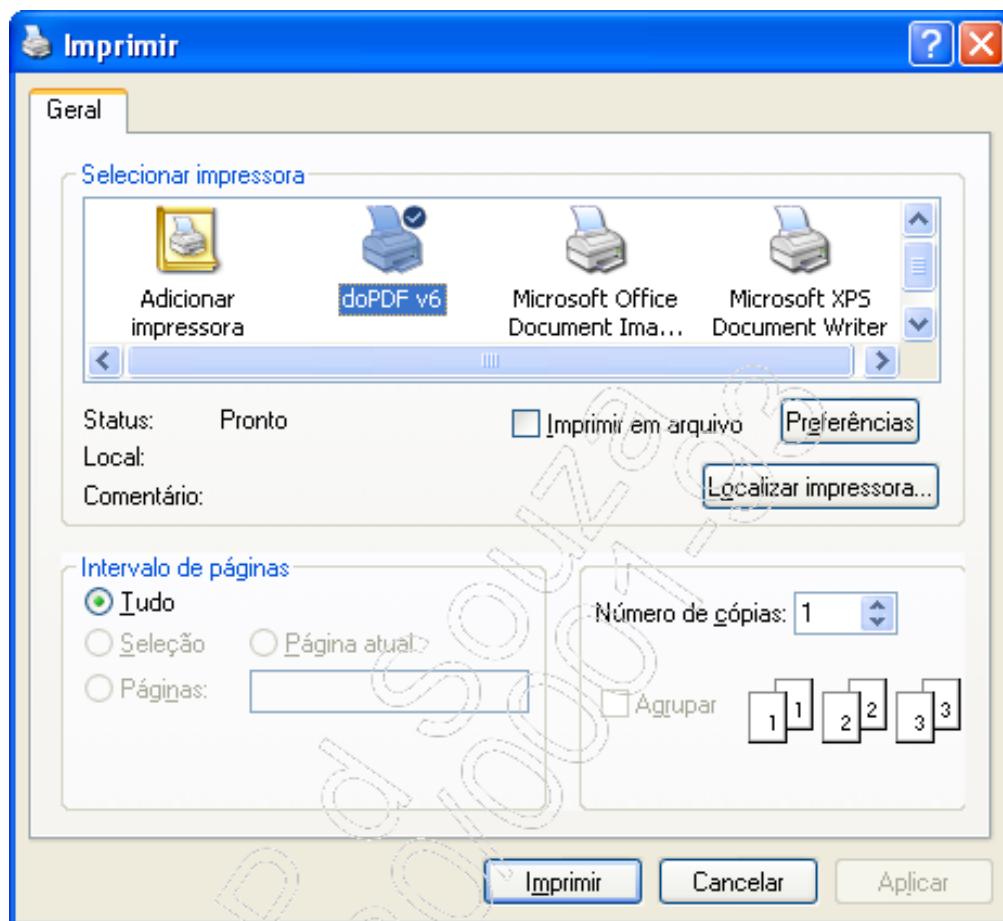
2.8. As caixas de diálogo padrão

Nas guias **Dialogs** e **Printing** da **Toolbox**, encontramos alguns controles de caixa de diálogo aptos a serem implementados na aplicação.

Para tarefas mais corriqueiras nas aplicações Windows, como abertura e fechamento de arquivos, os controles de caixa de diálogo do Visual Studio já se encontram prontos para uso e as interfaces já estão construídas conforme os padrões para os trabalhos mais comuns. Isso evita a necessidade de se criar caixas de diálogo personalizadas para a realização dessas tarefas. Torna-se necessário, em muitos casos, apenas escrever os procedimentos de evento que estabelecem a ligação entre as caixas de diálogo e o programa.

A seguir, descreveremos os controles de caixa de diálogo disponíveis:

- **PrintDialog**: Por meio desse controle, é possível configurar opções de impressão. A figura a seguir ilustra a caixa de diálogo **PrintDialog** aberta:

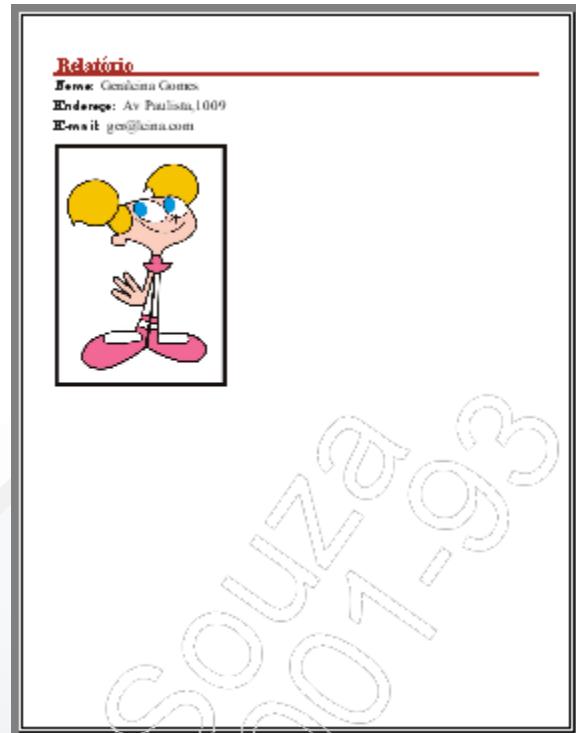


- **PrintDocument**: Por meio desse controle, é possível ajustar propriedades relacionadas à impressão do documento em aplicações baseadas em Windows. Para abranger todas as configurações relacionadas à impressão, pode ser utilizado em conjunto com o controle **PrintDialog**. A figura a seguir ilustra o controle **PrintDocument**:

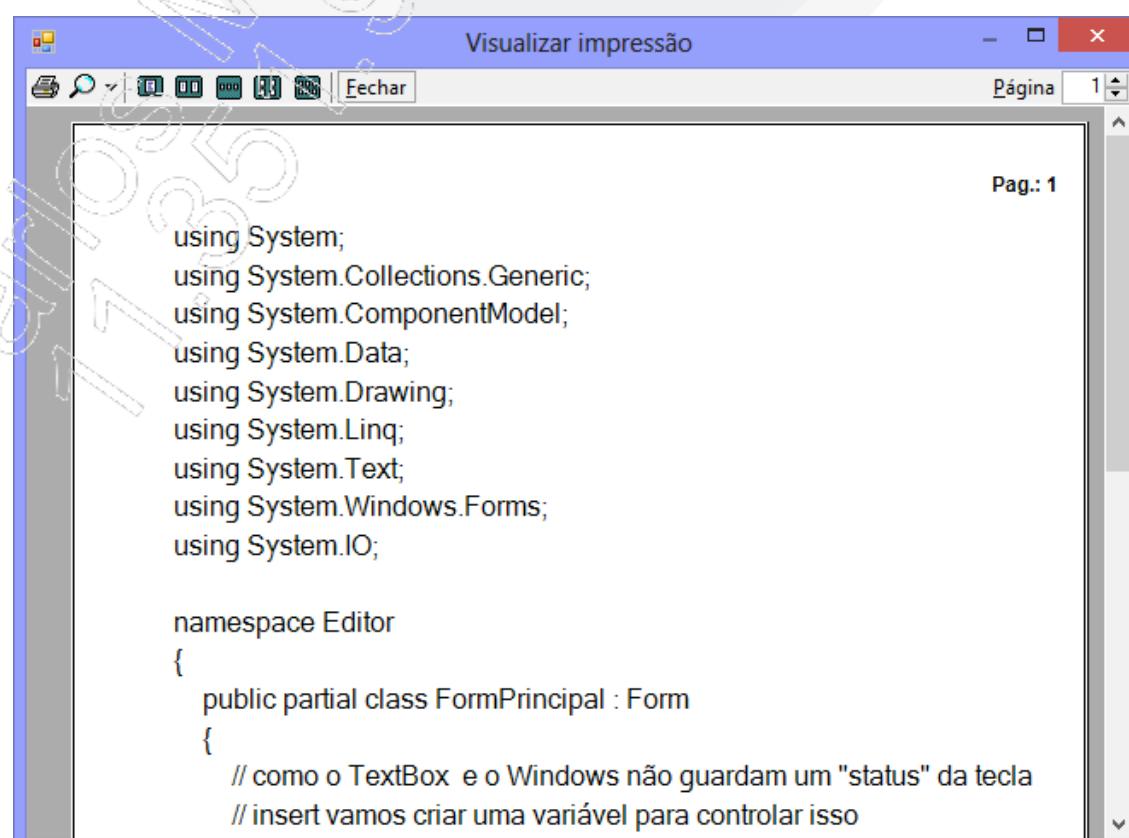


C# - Módulo I

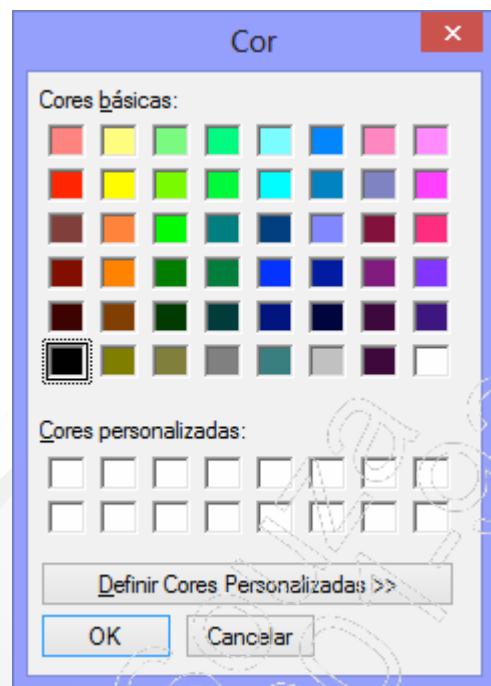
- **PrintPreviewControl**: Por meio desse controle, é possível obter a visualização do documento como será impresso. A caixa de diálogo, exibida a seguir, não oferece outras configurações, somente a possibilidade de visualizar a impressão:



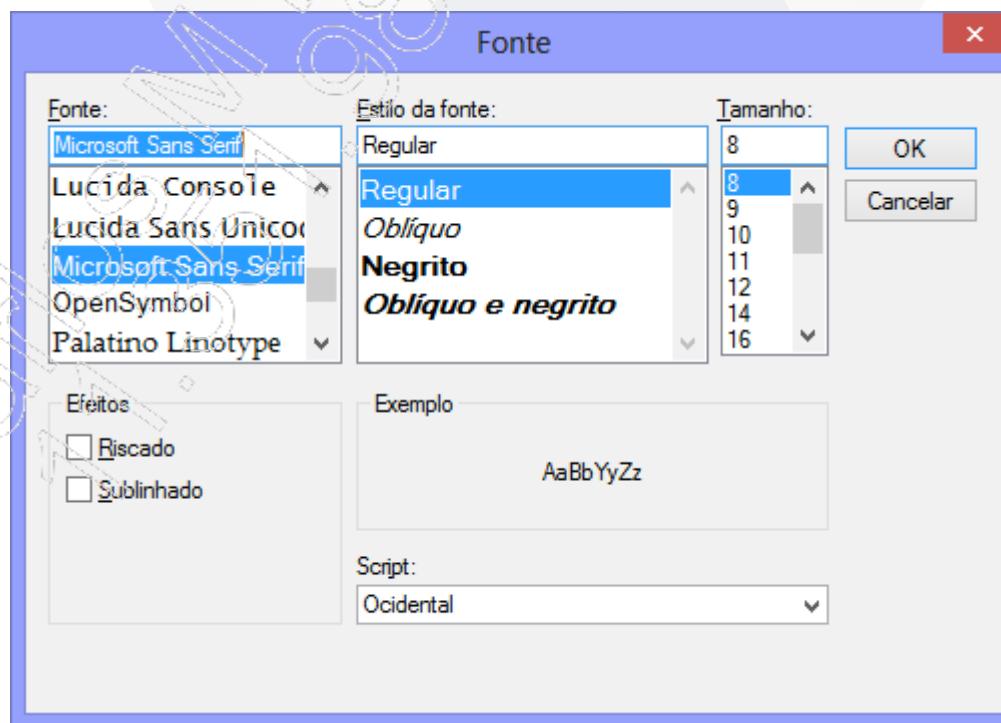
- **PrintPreviewDialog**: Por meio desse controle, é possível exibir uma caixa de diálogo para a apresentação prévia da impressão. A figura a seguir ilustra a caixa de diálogo **PrintPreviewDialog** aberta:



- **ColorDialog**: Esse controle possibilita escolher uma cor a partir de uma paleta de cores. A imagem a seguir ilustra a caixa de diálogo **ColorDialog** aberta:

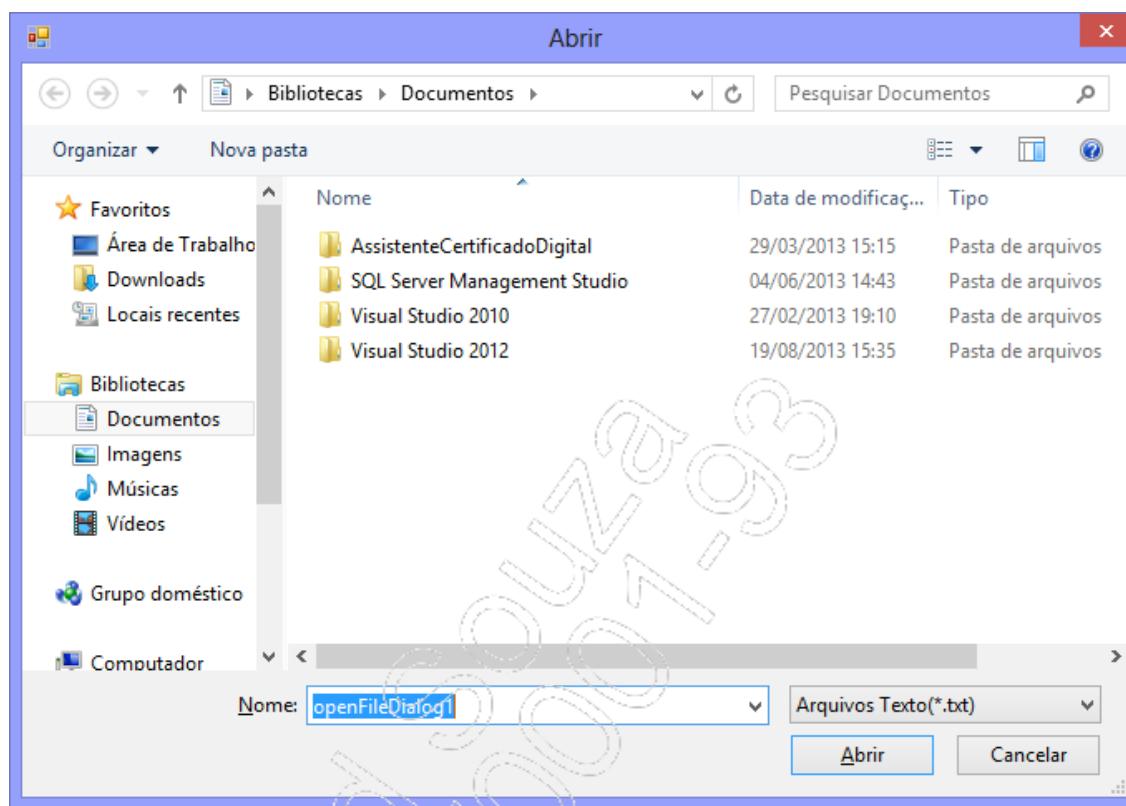


- **FontDialog**: Esse controle permite escolher um novo tipo de fonte e estilo. Vejamos, a seguir, a caixa de diálogo **FontDialog** aberta:

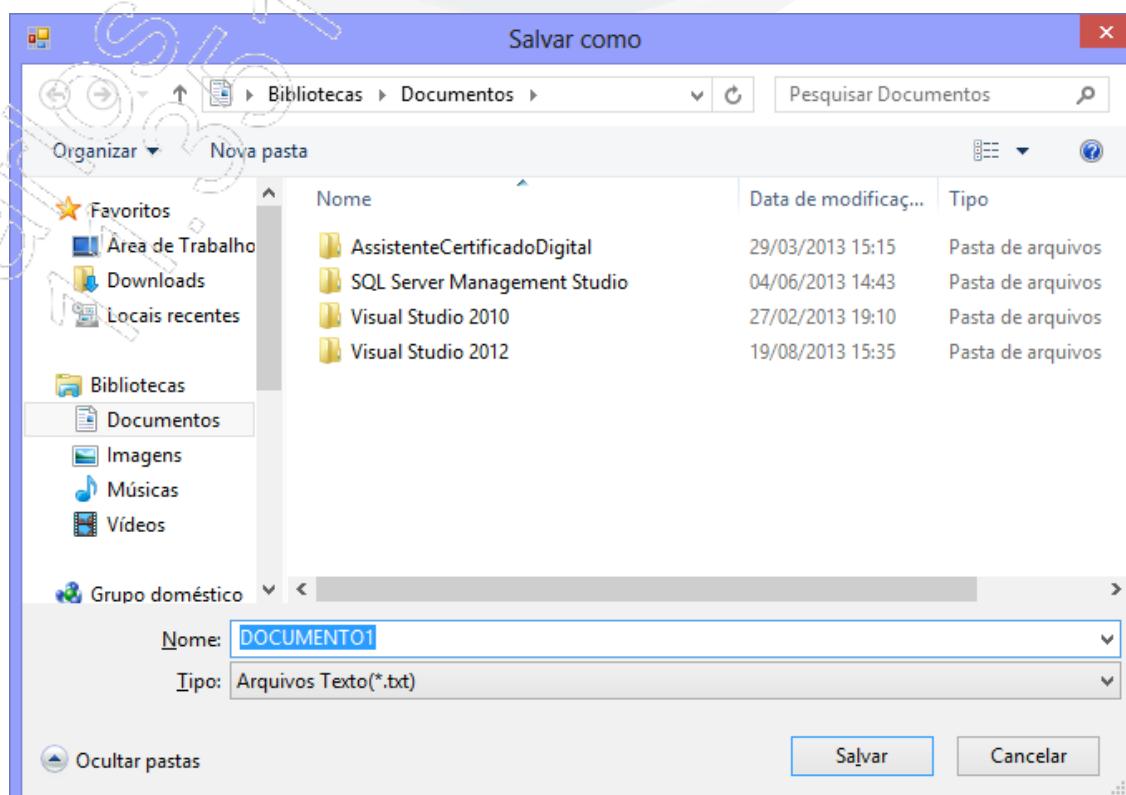


C# - Módulo I

- **OpenFileDialog:** Com a utilização desse controle, é possível indicar o drive, o nome da pasta e o nome para um arquivo já existente. A imagem a seguir ilustra a caixa de diálogo **OpenFileDialog** aberta:



- **SaveFileDialog:** Por meio desse controle, é possível indicar o drive, o nome da pasta e o nome para um novo arquivo. A imagem a seguir ilustra a caixa de diálogo **SaveFileDialog** aberta:



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- As interfaces de usuário são os projetos mais comuns criados por meio do Visual Studio;
- Os controles são selecionados a partir da janela **ToolBox** e utilizados para desenhar, com o mouse, objetos em um formulário;
- Podemos definir um objeto como sendo um elemento de interface do usuário criado em um formulário C# com um controle da **ToolBox**;
- As propriedades são responsáveis por manipular o comportamento dos controles. Podemos defini-las como sendo um valor ou uma característica mantida por um objeto;
- Uma procedure de evento é um bloco de código que entra em ação quando é associado a um evento disparado pelo objeto;
- Nas situações em que se torna necessário executar uma ação ou um serviço para um objeto específico em uma aplicação, podemos fazer uso das funcionalidades dos métodos;
- Enquanto o **MenuStrip** representa o contêiner da estrutura do menu de um formulário, os objetos **ToolStripMenuItem** representam os comandos contidos na estrutura do menu ou, ainda, um menu pai para outros itens;
- Depois de criar uma barra de ferramentas em uma aplicação, é necessário escrever procedures de evento para fazê-la funcionar, assim como ocorre nos comandos de menu;
- Existem duas maneiras de exibir um formulário: ou abrindo-o de modo exclusivo, ou seja, nenhuma outra janela fica disponível até que ele seja fechado; ou de modo compartilhado, situação em que ele fica disponível junto com outros formulários;
- Para tarefas mais corriqueiras nas aplicações Windows, como abertura e fechamento de arquivos, os controles de caixa de diálogo do Visual Studio já se encontram prontos para uso e as interfaces já estão construídas conforme os padrões para os trabalhos mais comuns.

2

Formulários

Teste seus conhecimentos

Carlos M. Souza
77.357.9007-003



IMPACTA
EDITORA

1. A partir de qual janela os controles são selecionados?

- a) ControlBox
- b) ToolBox
- c) Properties
- d) SolutionExplorer
- e) ServerExplorer

2. Qual a propriedade mais importante dos controles TextBox e Label?

- a) Font
- b) ForeColor
- c) BackColor
- d) Text
- e) Anchor

3. Qual o evento mais importante do controle Button?

- a) MouseEnter
- b) MouseLeave
- c) KeyPress
- d) Click
- e) KeyDown

4. Qual alternativa lista três propriedades importantes de um CheckBox?

- a) Text, Name, Checked.
- b) ForeColor, BackColor, Value.
- c) Text, Value, Maximum.
- d) Min, Max, Value.
- e) Minimum, Maximum, CheckedState.

5. Qual alternativa contém três propriedades importantes de um TextBox?

- a) Multiline, Checked, Value.
- b) Text, Name, Items.
- c) Minimum, Maximum, Value.
- d) ReadOnly, Text, Name.
- e) Value, PasswordChar, Hot-Track.

2

Formulários

Mãos à obra!

Carlos M. Souza
77.357.9007-003



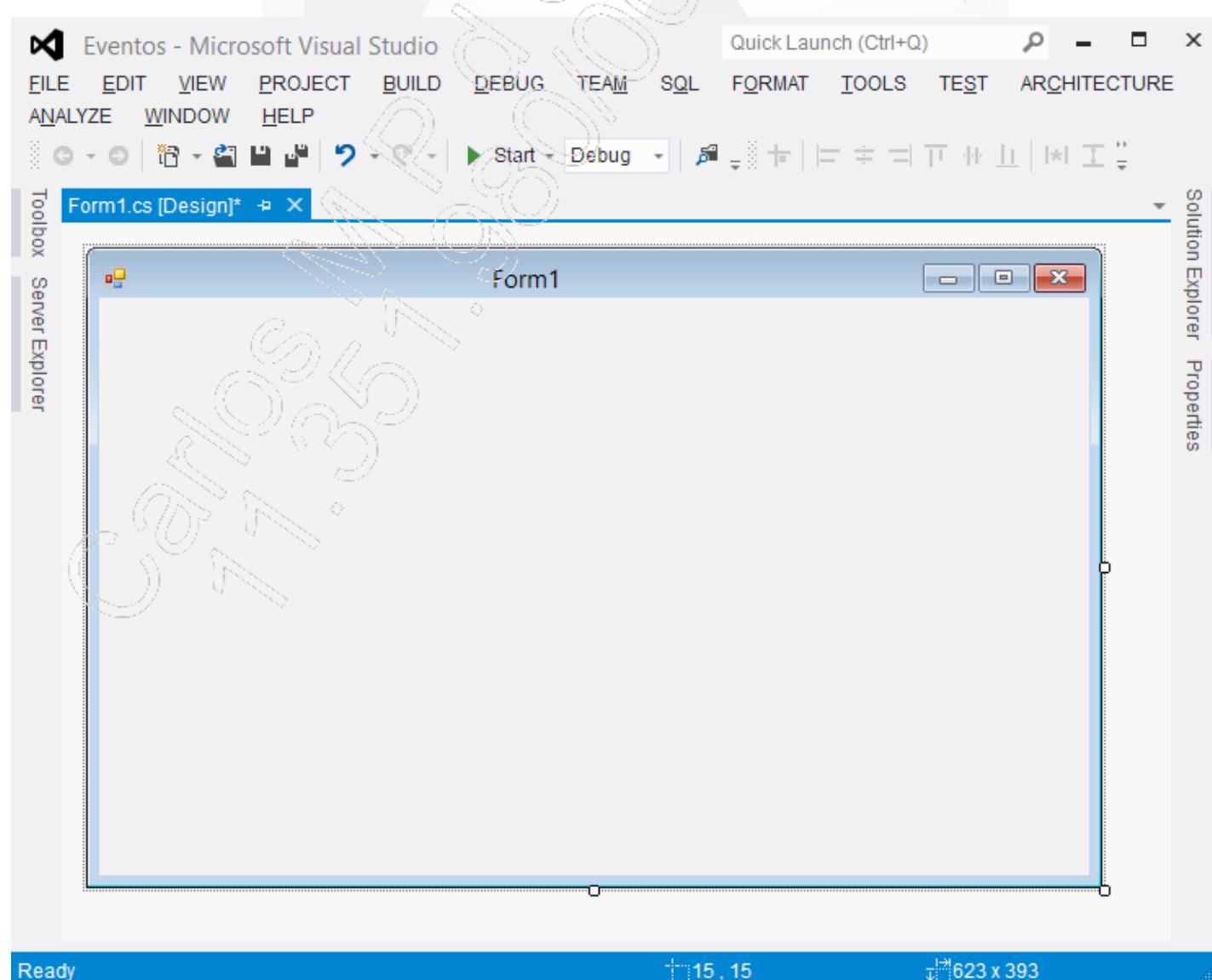
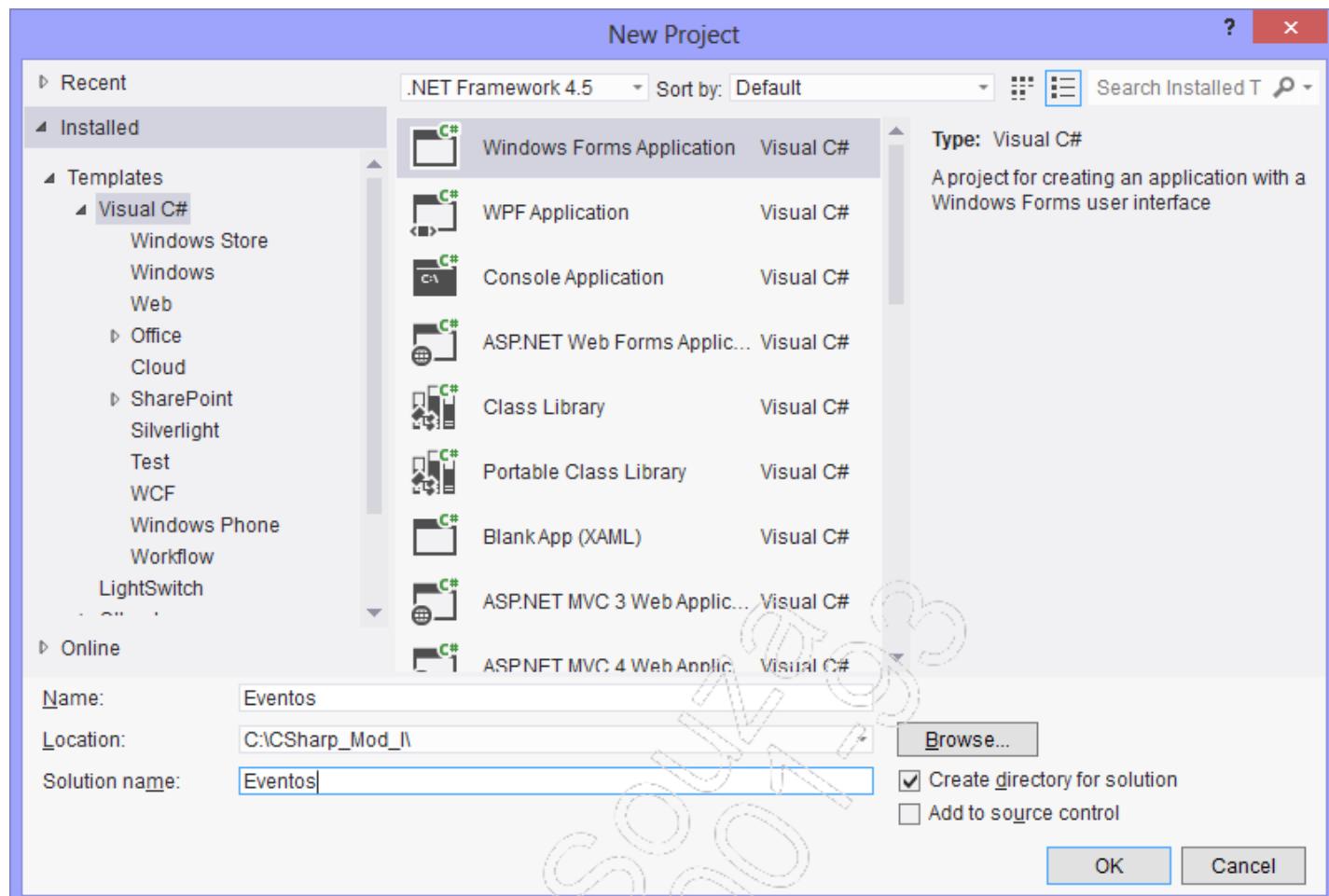
IMPACTA
EDITORA

Laboratório 1

A - Criando um novo projeto Windows Form para demonstrar alguns tipos de evento

1. Clique em **File**;
2. Aponte para **New**;
3. Clique em **Project**;
4. Escolha **Visual C#** na lista de tipos de projetos;
5. Escolha **Windows Forms Application** na lista de templates;
6. Digite um nome para o projeto na caixa de texto **Name: Eventos**;
7. Escolha o local em que será salvo o projeto. Este passo é opcional. Se o local não for escolhido, o projeto será, por padrão, salvo na pasta **Visual Studio 2012**, em **Meus Documentos**;
8. Para o sistema criar uma pasta para a solução, mantenha a opção **Create directory for solution** marcada;
9. Clique em **OK**. O resultado será o seguinte:

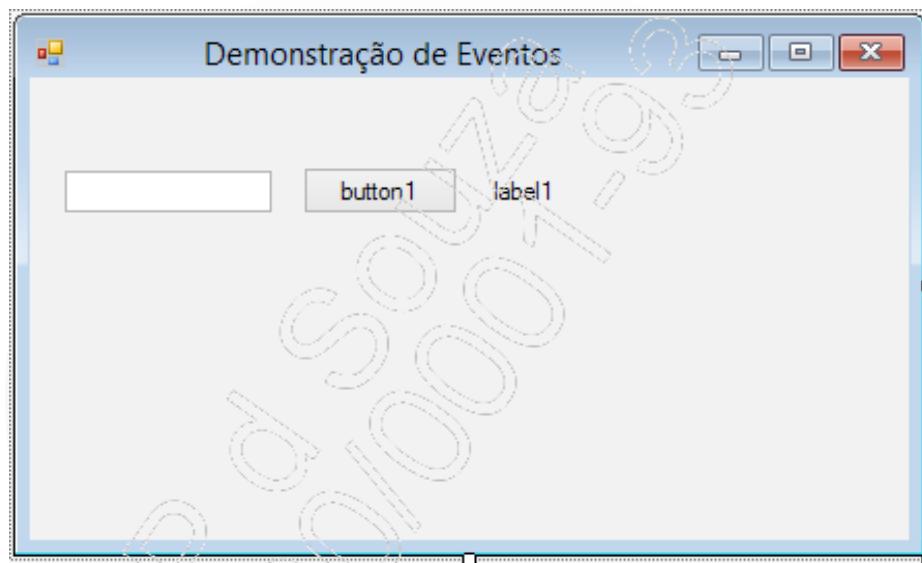
Um projeto sempre faz parte de uma solução, que pode possuir múltiplos projetos.



10. Altere a propriedade **Text** do formulário para **Demonstração de Eventos**;

11. Coloque sobre o **Form** os componentes **TextBox**, **Button** e **Label**:

- **TextBox**: Componente que permite digitação. A propriedade **Text** devolve o texto digitado;
- **Button**: Quando clicado, deverá copiar o texto digitado no **TextBox** para o **Label**;
- **Label**: Exibe um texto. A propriedade **Text** define o texto que será exibido.



12. Selecione o botão e procure o evento **Click**. Crie um método para este evento. Nele, a propriedade **Text** do **Label** deve receber o **Text** do **TextBox**;

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = textBox1.Text;
}
```

Sempre que criarmos um método a partir da janela de eventos, ele receberá dois parâmetros:

- **sender**: É o objeto que acionou o evento;
- **e**: Traz informações sobre o evento ocorrido.

Quando a variável **e** é do tipo **EventArgs**, ela não traz qualquer informação relevante sobre o evento.

13. Pressione a tecla F5 para executar o código;

14. Nesta próxima alteração, vamos emitir um beep para cada tecla pressionada em **textBox1**. Neste caso, usaremos o evento **KeyPress**, que é disparado sempre que pressionarmos uma tecla de texto;

A classe **Console** possui um método chamado **Beep** que emite um som no autofalante interno do computador:

Console.Beep(freqHertz, duracaoMiliSeg)

Em que:

- **freqHertz**: É a frequência do som em Hertz. Pode variar de 37 até 32767;
- **duracaoMiliSeg**: É a duração do som em milissegundos.

15. Selecione o **TextBox**, crie o método para o evento **KeyPress** e emita um som de frequência 500 hertz durante dois décimos de segundo;

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    // Complete você
}
```

16. Observe que, neste caso, o segundo parâmetro recebido pelo método (variável **e**) é do tipo **KeyPressEventArgs**, portanto, ela traz informações sobre o evento;

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
```

The image shows a screenshot of a C# code editor. A tooltip is displayed over the variable 'e' in the 'KeyPressEventArgs' type. The tooltip contains the following information:

char KeyPressEventArgs.KeyChar
Gets or sets the character corresponding to the key pressed.

C# - Módulo I

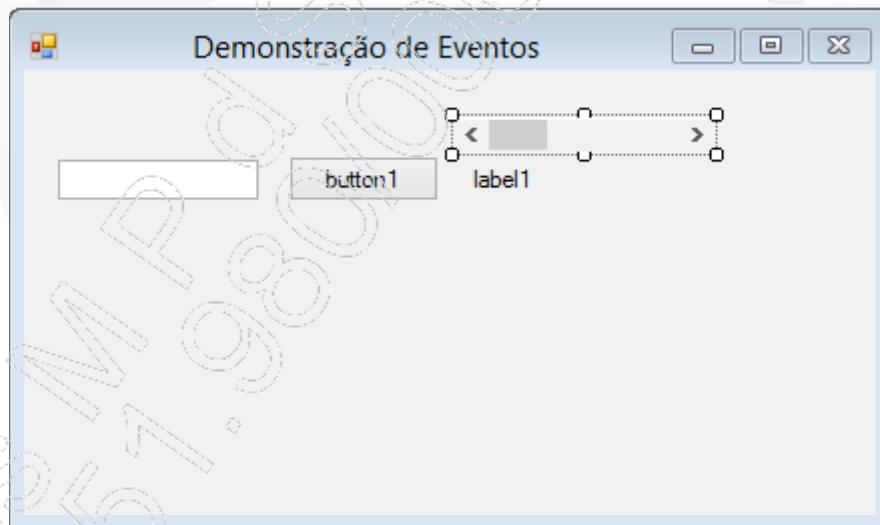
17. A propriedade **KeyChar** permite ler ou alterar o caractere correspondente à tecla pressionada. Apenas para fazer um teste, inclua a seguinte instrução na última linha do método:

```
e.KeyChar = 'X';
```

18. Veremos que qualquer tecla pressionada se transformará em X. Podemos também mostrar o código da tecla pressionada na barra de título do formulário. Comente a linha anterior e coloque esta outra:

```
// e.KeyChar = 'X';
Text = ((int)e.KeyChar).ToString();
```

19. Traga agora um componente **HScrollBar** e posicione-o de acordo com a imagem a seguir. Esta ScrollBar servirá para alterar o tamanho da letra do Label. O tamanho indicado poderá variar de 8 a 40;



20. Altere as seguintes propriedades:

- **Name:** fonteHScrollBar;
- **Minimum:** 8;
- **Maximum:** 40.

21. Uma **ScrollBar** possui também uma propriedade chamada **Value**, que mostra o valor atual indicado por ela. Esta propriedade é alterada toda vez que movermos o indicador de posição da **ScrollBar**. Seus principais eventos são:

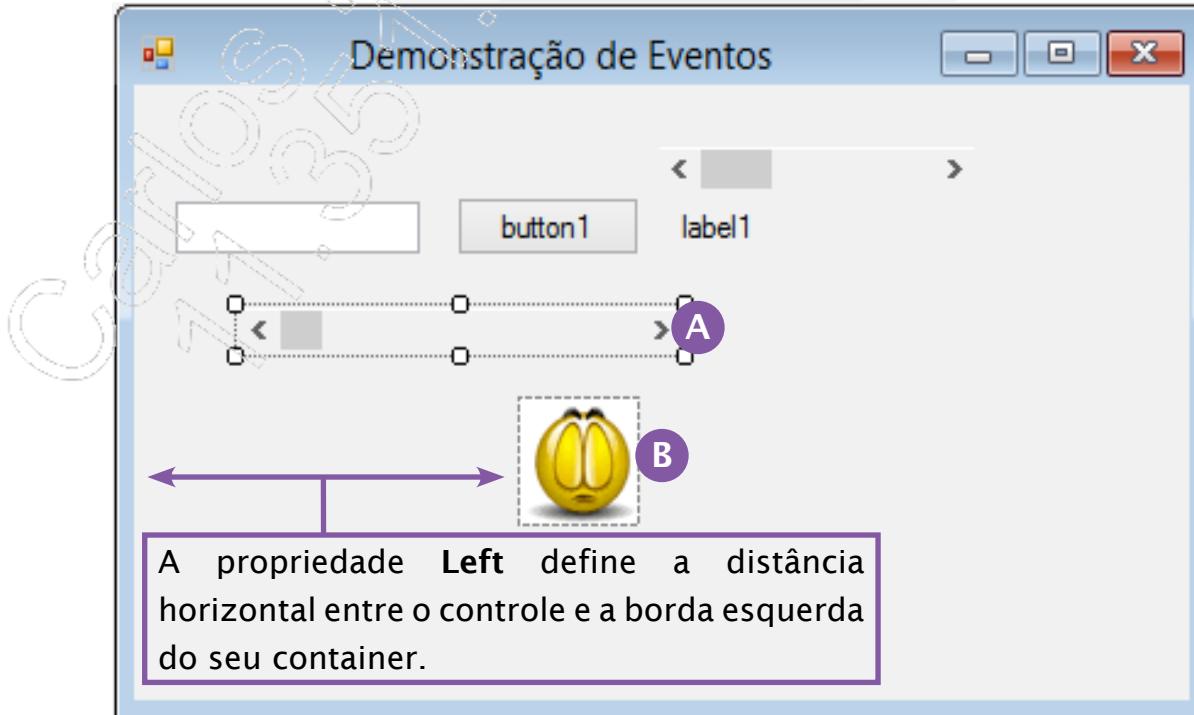
- **Scroll**: Executado quando o usuário age sobre a **ScrollBar**, arrastando ou clicando;
- **ValueChanged**: Executado sempre que a propriedade **Value** sofrer alteração, seja por ação do usuário ou via programação.

22. Crie um método para o evento **ValueChanged** da **ScrollBar**:

```
private void fonteHScrollBar_ValueChanged(object sender, EventArgs e)
{
    // substitui a fonte atual do label por uma nova fonte, do mesmo
    // tipo, mas com o tamanho definido pela ScrollBar
    label1.Font = new Font(label1.Font.Name, fonteHScrollBar.Value);
}
```

23. Execute o código até este ponto pressionando F5;

24. Coloque no formulário os componentes mostrados a seguir. O objetivo é mover o objeto **PictureBox** usando a **ScrollBar**:



- **A - HScrollBar**
 - **Name:** moveHScrollBar.
- **B - PictureBox**
 - **Name:** moveHPictureBox;
 - **Image:** Clique nos pontinhos da propriedade **Image**. Na tela seguinte, clique no botão **Import**. Localize a imagem presente no exemplo anterior na pasta **1_Imagens** fornecida pelo instrutor.

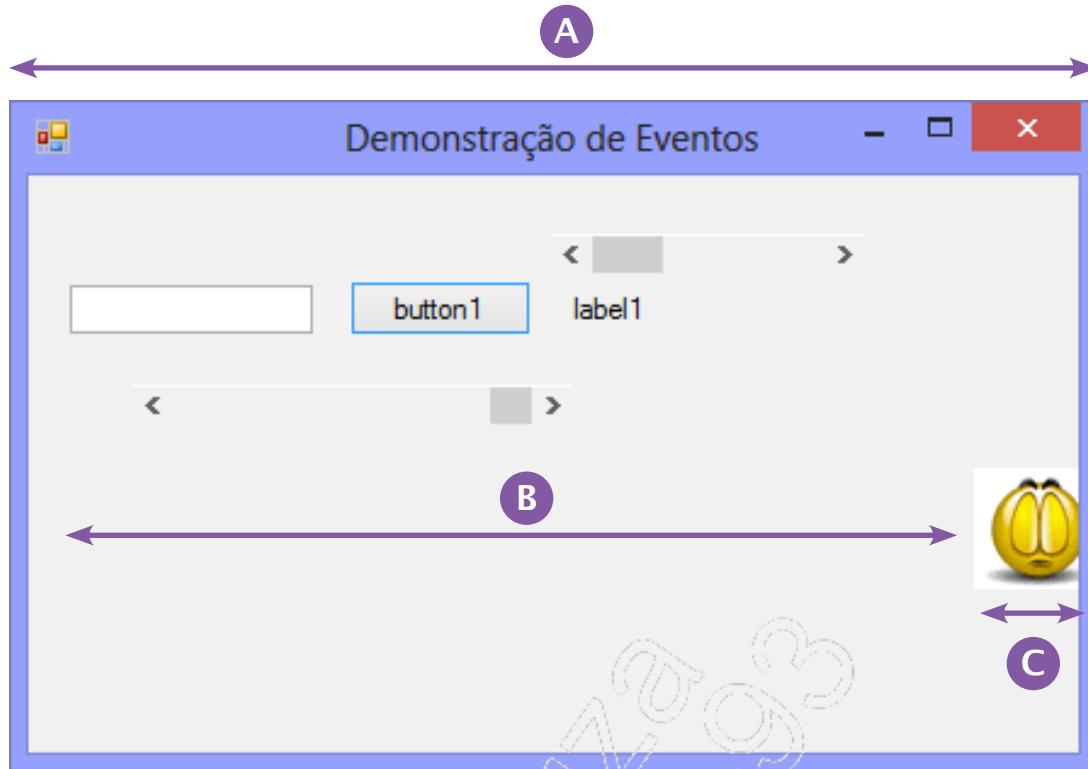
Como podemos deduzir pelo exemplo anterior, quanto maior o valor da propriedade **Left**, mais à direita da tela estará o controle.

25. Utilize o evento **ValueChanged** de **moveHScrollBar**:

```
private void moveHScrollBar1_ValueChanged(object sender, EventArgs e)
{
    moveHPictureBox.Left = moveHScrollBar1.Value;
}
```

26. Pressione F5 para executar esse código;

Note que, na imagem anterior, quando a ScrollBar atinge o seu valor máximo, a figura não passa nem da metade do formulário. Que alteração devemos fazer para que a figura chegue até o final do formulário?



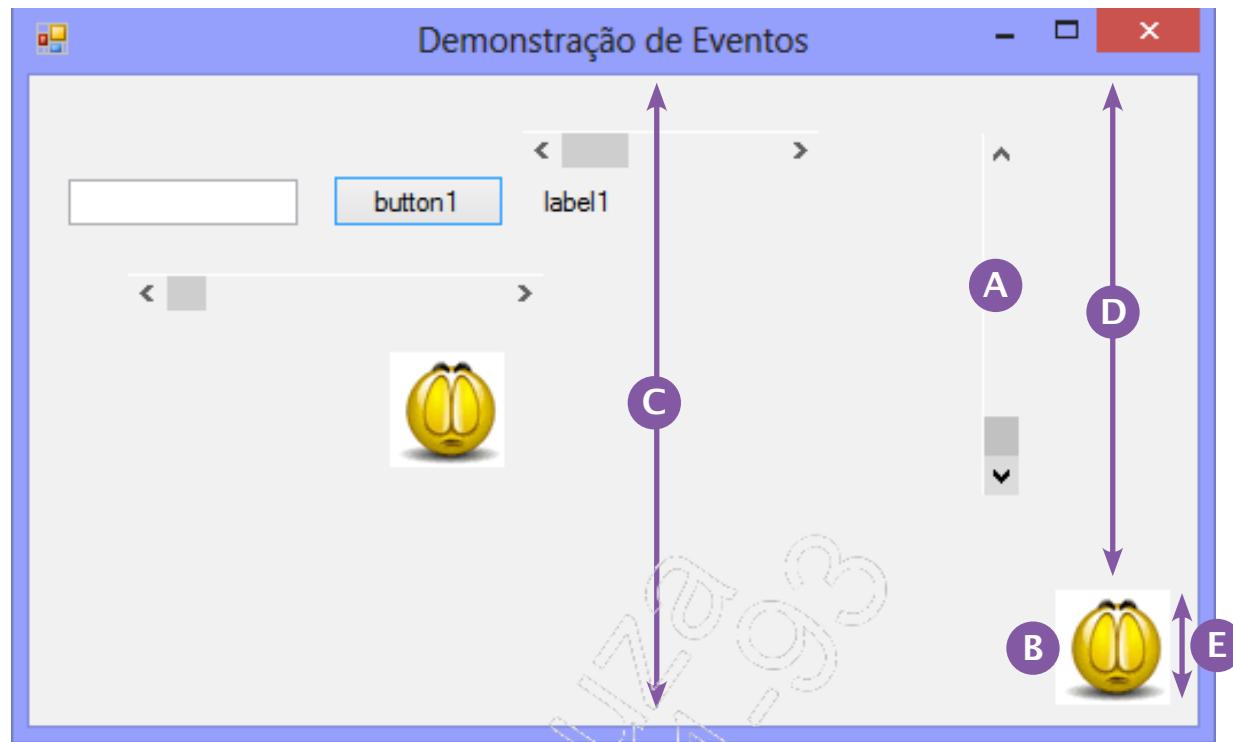
- A - **this.Width;**
- B - **moveHPictureBox.Left;**
- C - **moveHPictureBox.Width.**

27. A propriedade **Maximum** de **moveHScrollBar** precisa ser calculada da seguinte forma:

```
moveHScrollBar.Maximum = this.Width - moveHPictureBox.Width
```

28. Inclua essa instrução no início do evento **ValueChanged** da ScrollBar;

29. Inclua os seguintes controles:



- **A - VScrollBar:** ScrollBar vertical.
 - Name: moveVScrollBar.
- **B - PictureBox:**
 - Name: moveVPictureBox.
- **C - this.ClientSize.Height**
- **D - moveVPictureBox.Top**
- **E - moveVPictureBox.Height**

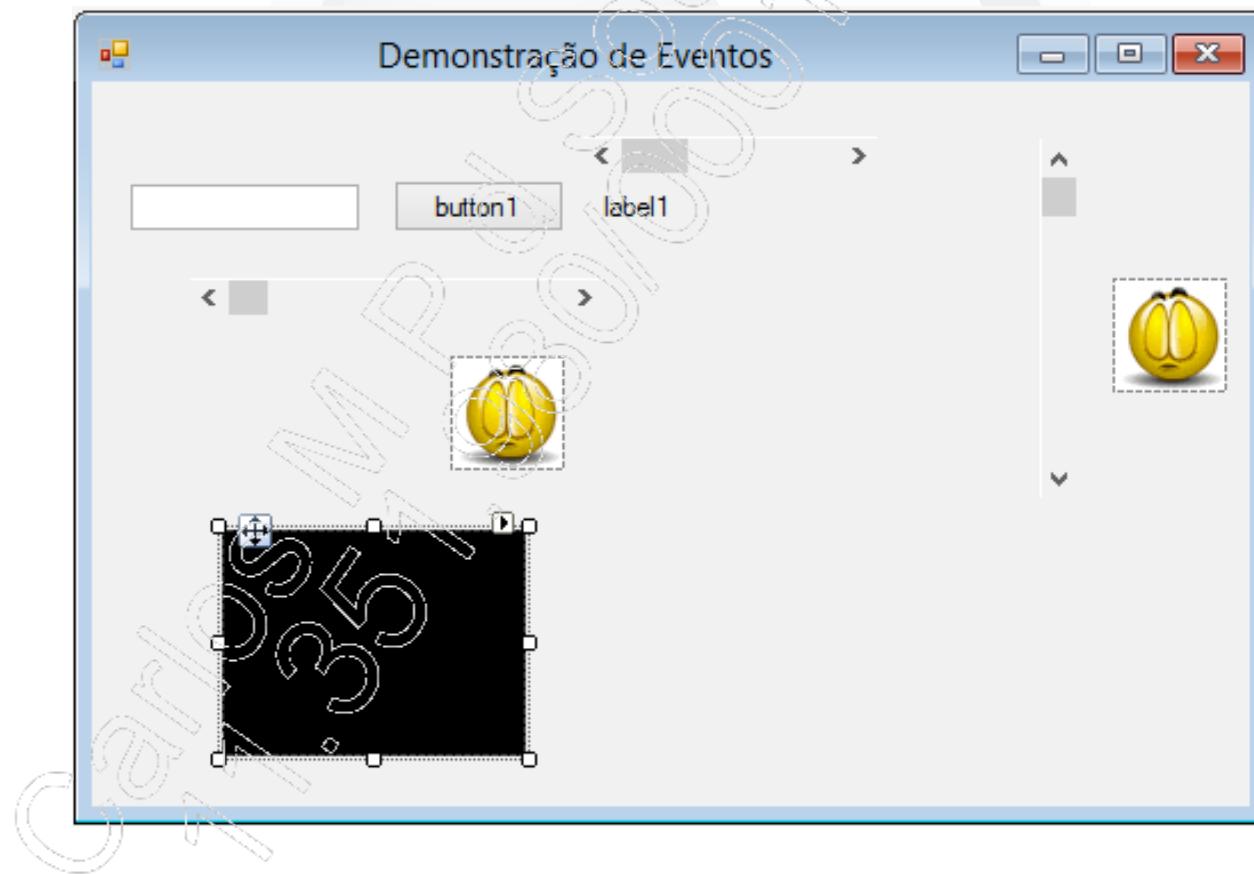
30. Crie um método para o evento **ValueChanged** de **moveVScrollBar** para mover o **PictureBox** verticalmente em toda a extensão do formulário;

```
private void moveVScrollBar_ValueChanged(object sender, EventArgs e)
{
    // configurar o valor máximo da ScrollBar de acordo com a altura
    // do formulário (complete)

    // posicionar o PictureBox de acordo com Value da ScrollBar
    // (complete)

}
```

31. Insira um componente **Panel** no formulário, de acordo com a imagem a seguir:



- **Name:** corPanel;
- **BackColor:** Preto.

C# - Módulo I

32. Quando o mouse entrar na área do Panel (evento **MouseEnter**), sua cor de fundo (**BackColor**) deve mudar para vermelho. Agora, quando o mouse sair (evento **MouseLeave**), o Panel deve voltar para preto. Para mudar a cor de fundo de um controle, execute esta sintaxe no código adiante:

```
nomeControle.BackColor = Color.corDesejada;

private void lblCor_MouseEnter(object sender, EventArgs e)
{
    // complete

}

private void lblCor_MouseLeave(object sender, EventArgs e)
{
    // complete
}
```

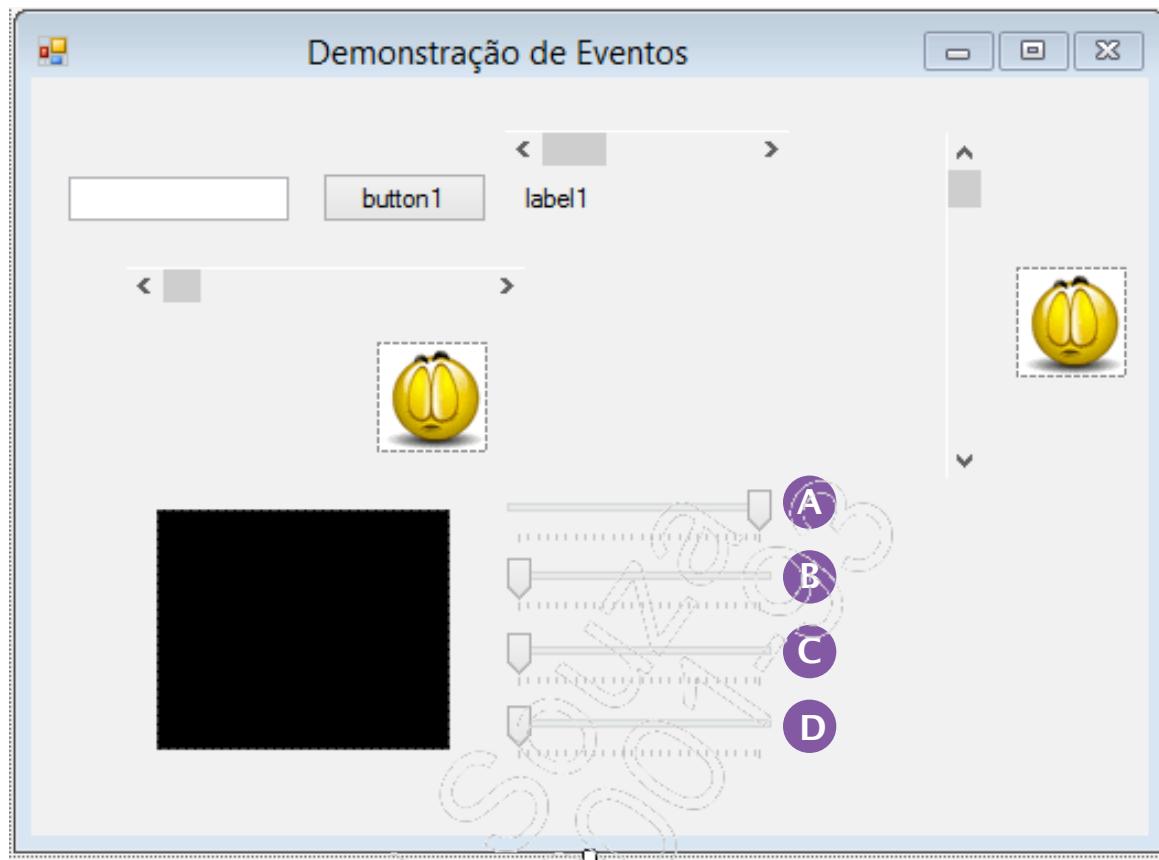
33. Para definir uma cor, são necessárias quatro informações:

- Grau de transparência (alpha). Pode variar de zero (totalmente transparente) até 255 (totalmente opaco);
- Quantidade de vermelho (red). Pode variar de zero (nada) a 255 (tudo);
- Quantidade de verde (green). Pode variar de zero (nada) a 255 (tudo);
- Quantidade de azul (blue). Pode variar de zero (nada) a 255 (tudo).

34. Misturando essas quatro informações, podemos formar mais de 16 milhões de cores, sendo cada uma delas com 256 níveis de opacidade diferentes. Para isso, usaremos:

```
Color cor = Color.FromArgb( alpha, red, green, blue)
```

35. Para testar esse recurso, altere o formulário colocando nele quatro componentes **TrackBar**, de acordo com a imagem a seguir:



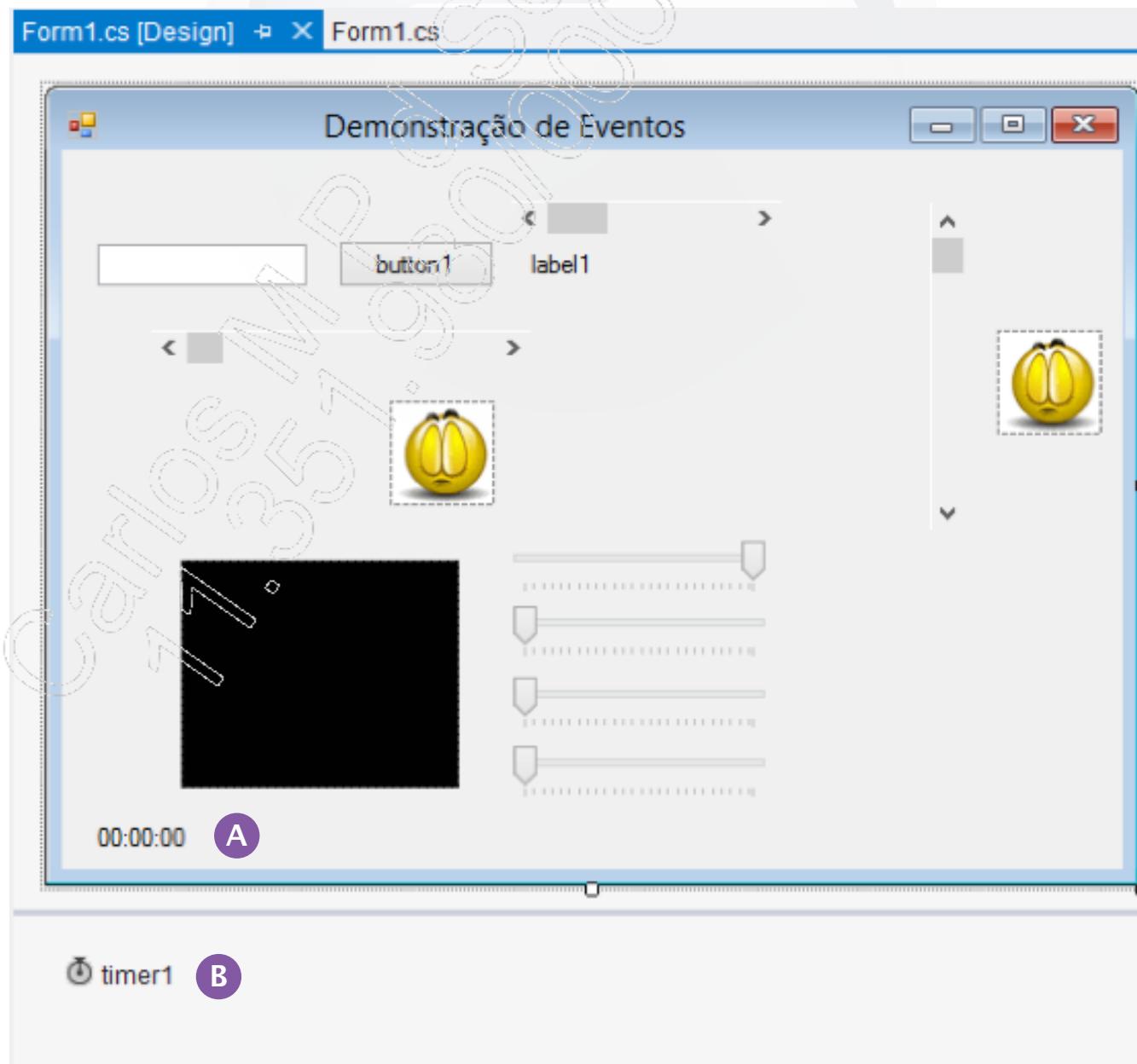
36. Altere a propriedade **Maximum** de todas elas para 255:

- A - Name: alphaTrackBar; Value: 255;
- B - Name: redTrackBar; Value: 0;
- C - Name: greenTrackBar; Value: 0;
- D - Name: blueTrackBar; Value: 0.

37. Neste caso, as quatro TrackBars vão executar exatamente as mesmas instruções. Então selecione as quatro e faça um único evento **ValueChanged** para todas elas:

```
private void blueTrackBar_ValueChanged(object sender, EventArgs e)
{
    corPanel.BackColor = Color.FromArgb(alphaTrackBar.Value,
                                         redTrackBar.Value,
                                         greenTrackBar.Value,
                                         blueTrackBar.Value);
}
```

38. Traga para o formulário um componente **Label** e um **Timer**. O Timer faz parte de um grupo de componentes chamados “não visuais”. Ele não aparece quando executamos a aplicação ou no design do formulário, apenas é mostrado na parte inferior da tela, fora do formulário;



- A - Label

- **Name:** relogioLabel;
- **Text:** 00:00:00.

- B - Timer

- **Enabled:** true (ligado);
- **Interval:** 1000 (1 segundo).

39. Selecione a janela de eventos do Timer e crie um método para o evento **Tick**, que é executado a cada intervalo de tempo configurado na propriedade **Interval**.

```
private void timer1_Tick(object sender, EventArgs e)
{
    relogioLabel.Text = DateTime.Now.ToString("hh:mm:ss");
    /* Opções para formatação de Data/Hora
     * d: dia com 1 ou 2 algarismos
     * dd: dia com 2 algarismos
     * M: mês com 1 ou 2 algarismos
     * MM: mês com 2 algarismos
     * yy: ano com 2 dígitos
     * yyyy: ano com 4 dígitos
     *
     * ddd: abreviação do nome do dia da semana
     * dddd: nome do dia da semana
     * MMM: abreviação do nome do mês
     * MMMM: nome do mês
     *
     * hh: hora de 0 a 12
     * HH: hora de 0 a 23
     * mm: minuto
     * ss: segundo
     * fff: milissegundos
     */
}
```


Instruções, tipos de dados, variáveis e operadores

3

- ✓ Instruções;
- ✓ Tipos de dados;
- ✓ Variáveis;
- ✓ Operadores.

Carloso
77.320-0007-03
SOUZA



IMPACTA
EDITORA

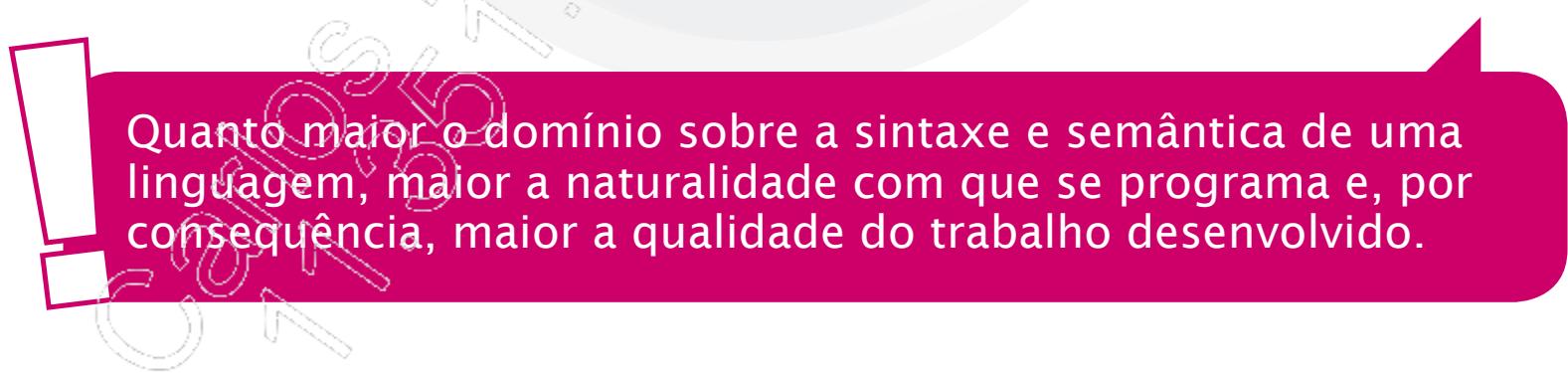
3.1. Introdução

Para começar a programar, é necessário primeiro conhecer os elementos constituintes da sintaxe e semântica do Visual C#, como instruções, identificadores e palavras-chave. Também devemos conhecer os tipos de dados existentes e as características principais dos valores que eles armazenam, as variáveis e os operadores aritméticos que nos permitem manipular valores e criar expressões.

3.2. Instruções

Para iniciarmos os estudos relativos às instruções de C#, devemos antes conhecer os conceitos a seguir:

- Instrução é um comando responsável pela execução de uma determinada ação;
- Método é uma sequência nomeada de instruções;
- Sintaxe é o conjunto de instruções que obedecem a uma série de regras específicas para seu formato e construção;
- Semântica é a especificação do que as instruções realizam.



Quanto maior o domínio sobre a sintaxe e semântica de uma linguagem, maior a naturalidade com que se programa e, por consequência, maior a qualidade do trabalho desenvolvido.

3.2.1. Identificadores

Os elementos nos programas, para serem distinguidos, são representados por nomes que chamamos de identificadores. A criação de identificadores deve seguir duas regras simples. A primeira é a obrigatoriedade de se utilizar apenas letras, números inteiros de 0 a 9 e o caractere sublinhado (_); e a segunda é que o primeiro elemento de um identificador deve ser sempre uma letra ou o sublinhado, mas nunca um número. Assim, **valortotal**, **_valor** e **preçoDeCusto** são identificadores válidos. Já **valortotal\$**, **\$valor** e **1preçoDeCusto** são inválidos.

Além dessas regras, devemos também lembrar que os identificadores fazem distinção entre letras maiúsculas e minúsculas, de modo que **valortotal** e **valorTotal** são identificadores distintos. Embora acentos sejam permitidos, evite utilizá-los.

3.2.1.1. Palavras reservadas

Palavras-chave são identificadores exclusivos da linguagem C# que não podem ser atribuídos a outras finalidades. As palavras-chave somam um total de 77 identificadores com significados específicos. Podemos reconhecer uma palavra-chave na janela **Code and Text Editor** por sua cor azul (esta cor é configurável dentro do menu **Tools / Options**).

abstract	decimal	float	namespace	return	try
as	default	for	new	sbyte	typeof
base	delegate	foreach	null	sealed	uint
bool	do	goto	object	short	ulong
break	double	if	operator	sizeof	unchecked
byte	else	implicit	out	stackalloc	unsafe
case	enum	in	override	static	ushort
catch	event	int	params	string	using
char	explicit	interface	private	struct	virtual
checked	extern	internal	protected	switch	void
class	false	is	public	this	volatile
const	finally	lock	readonly	throw	while
continue	fixed	long	ref	true	

Existem também outros identificadores que não são exclusivos de C# e que podem ter seu significado alterado, embora isso não seja aconselhável. São eles:

dynamic	group	let	select	var
from	into	orderby	set	where
get	join	partial	value	yield

3.3. Tipos de dados

Em C#, qualquer variável que vá ser usada dentro de um procedimento precisa, antes de tudo, ter seu tipo definido.

A seguir, temos uma tabela com diversos tipos predefinidos de C#, chamados de tipos de dados primitivos:

- **Tipos inteiros**

Tipo/Tipo .Net	Faixa	Bytes
sbyte/SByte	-128 a 127	Com sinal 8-bit.
byte/Byte	0 a 255	Sem sinal 8-bit.
char/Char	U+0000 a U+ffff	Unicode 16-bit character.
short/Int16	-32,768 a 32,767	Com sinal 16-bit.
ushort/UInt16	0 a 65,535	Sem sinal 16-bit.
int/Int32	-2,147,483,648 a 2,147,483,647	Com sinal 32-bit.
uint/UInt32	0 a 4,294,967,295	Sem sinal 32-bit.
long/Int64	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	Com sinal 64-bit.
ulong/UInt64	0 a 18,446,744,073,709,551,615	Sem sinal 64-bit.

- Tipos com ponto flutuante

Tipo/ Tipo .Net	Faixa	Precisão
float/Single	$\pm 1.5\text{e}-45$ a $\pm 3.4\text{e}38$	7 dígitos.
double/Double	$\pm 5.0\text{e}-324$ a $\pm 1.7\text{e}308$	15-16 dígitos.
decimal/Decimal	$\pm 1.0 \times 10\text{e}-28$ a $\pm 7.9 \times 10\text{e}28$	28-29 dígitos.

- Outros

Tipo/Tipo .Net	Descrição
string/String	Texto Unicode.
bool/Boolean	true ou false (George Boole - 1815 a 1864).
object/Object	Qualquer tipo de objeto do .Net (semelhante a VARIANT) de outras linguagens.

3.4. Variáveis

As variáveis podem ser definidas como locais destinados ao armazenamento temporário de informações de diferentes tipos, como números, palavras, propriedades, datas, entre outras.

Para compreendermos a funcionalidade das variáveis, especialmente dos tipos de valores que elas podem conter, podemos considerar valores que surgem como resultado de cálculos específicos, ou mesmo aqueles provenientes da entrada de informações por parte de usuários no tempo de execução. As variáveis podem conter, ainda, o trecho de um dado que deve ser exibido no formulário.

3.4.1. Convenções

Ainda em relação às variáveis, é importante ressaltar que elas devem receber um nome único, o qual será utilizado para fazer referência ao valor por elas armazenado. Há certas convenções para nomeação de variáveis bastante úteis àqueles que pretendem evitar confusões, especialmente porque nomeá-las é uma tarefa delicada, que exige nomes curtos, intuitivos e de fácil memorização.

C# - Módulo I

A seguir, apresentamos algumas convenções importantes:

- Não utilizar sublinhados e não criar variáveis que diferem umas das outras apenas por meio de letras maiúsculas e minúsculas. Essa situação pode gerar confusões devido à semelhança dos nomes – por exemplo, uma variável chamada **VariavelTeste** e outra denominada **variavelTeste**;
- Iniciar o nome da variável com letra minúscula. Nos casos de variáveis com mais de uma palavra, empregar maiúscula no início da segunda palavra e nas posteriores. Esse procedimento é conhecido como notação **camelCase**;
- Não utilizar notação húngara, a qual consiste na especificação de prefixos nos nomes das variáveis para facilitar a identificação de seu tipo.

É importante salientar que, além de gerar confusões devido à semelhança dos nomes, a criação de variáveis que se diferem umas das outras apenas por meio de letras maiúsculas e minúsculas pode restringir a capacidade de reutilização de classes. Isso ocorre, especificamente, em aplicações desenvolvidas em linguagens como o Visual Basic, que não são case-sensitive, ou seja, não distinguem maiúsculas de minúsculas.

Devemos levar em consideração que a primeira recomendação apresentada – referente à importância de não utilizar sublinhados e não criar variáveis que se diferem umas das outras apenas por meio de letras maiúsculas e minúsculas – deve ser seguida por todos aqueles que pretendem escrever programas interoperáveis com outras linguagens, uma vez que se trata de uma recomendação em conformidade com a CLS (Common Language Specification).

3.4.2. Declaração de variáveis

Para declarar uma variável, devemos colocar o seu tipo e, em seguida, o nome da variável. Opcionalmente, podemos também atribuir valor a ela.

```
int idade;
string nome = "CARLOS MAGNO";
float salario = 25.5f;
double preco = 10.32;
decimal valor = 21.3m;
bool casado = false;
char sexo = 'F';
```

3.4.2.1. Variáveis locais de tipo implícito

Como vimos, quando atribuímos um valor a uma variável, ambos devem ser do mesmo tipo. Uma característica que o compilador do C# tem é a capacidade de verificar rapidamente qual o tipo de uma expressão que utilizamos para inicializar uma variável e informar se há correspondência entre o tipo da expressão e o da variável.

Conforme mostra o exemplo a seguir, podemos utilizar a palavra-chave **var** para que o compilador do C# deduza qual o tipo de uma determinada variável a partir do tipo da expressão que foi utilizada para inicializá-la. Então, podemos declarar a variável como sendo desse tipo. Vejamos:

```
private void button1_Click(object sender, EventArgs e)
{
    var x = 10;
    label1.Text = x.ToString();
    (local variable) int x

    var texto = "ooo";
    label1.Text = texto;
    (local variable) string texto

    var salario = 1234.56m;
    label1.Text = salario.ToString();
    (local variable) decimal salario
}
```

- Abrangência das variáveis

No editor de textos do VS2012, ao colocarmos o ponteiro do mouse sobre uma variável, ele nos informa a origem desta variável.

```
public partial class Form1 : Form
{
    // variável declarada para a classe, visível em todos os métodos
    decimal a = 10;

    public Form1()
    {
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        // se refere a variável da classe, declarada fora de método
        listBox1.Items.Add(a);
    }

    private void button2_Click(object sender, EventArgs e)
    {
        // Ambiguidade. O método declara variável de mesmo nome
        string a = "IMPACTA";
        // se refere a variável local, declarada dentro do método
        listBox1.Items.Add(a);
        (local variable) string a

        // se refere a variável da classe, declarada fora de método
        listBox1.Items.Add(this.a);
    }
}
```

3.5. Operadores

Os operadores indicam o tipo de operação matemática que será executado, gerando novos valores a partir de um ou mais operandos (os itens que ficam à direita ou à esquerda do operador e que são manipulados por ele). Normalmente, o resultado é do tipo booleano ou numérico.

3.5.1. Operador de atribuição

O sinal de igualdade (`=`) representa a atribuição de um valor a uma variável, em que a variável e o valor atribuído devem ser de tipos compatíveis, ou seja, uma variável do tipo **char** não pode receber um valor do tipo **boolean** (**true** ou **false**).

É possível atribuirmos uma variável primitiva a outra variável primitiva. Vejamos como isso ocorre no exemplo a seguir:

```
int a = 3;
int b = a;
```

Em que:

- **a** recebe o valor **3**;
- **b** recebe a variável **a**, logo **b** contém o valor **3**.

Nesse momento, as duas variáveis (**a**, **b**) têm o mesmo valor, porém, se alterarmos o valor de uma delas, a outra não será alterada. Vejamos o exemplo seguinte:

Operador	Descrição	Precedência
<code>=</code>	Atribuição simples	25
<code>+=</code>	Atribuição por adição	26
<code>-=</code>	Atribuição por subtração	26
<code>*=</code>	Atribuição por multiplicação	27
<code>/=</code>	Atribuição por divisão	27

```
// declara a variável a e lhe atribui o valor 10
int a = 10;
// soma 20 ao conteúdo atual da variável a, o que resulta 30
a += 20;
// subtrai 5 da variável a, o que resulta 25
a -= 5;
// multiplica a variável a por 2, o que resulta 50
a *= 2;
// divide o conteúdo da variável a por 5, o que resulta 10
a /= 5;
```

3.5.2. Operadores aritméticos

Os operadores aritméticos descritos na tabela adiante são utilizados nos cálculos matemáticos:

Operador	Descrição	Tipo	Precedência
<code>++</code>	Pós-incremento	Unário	1
<code>--</code>	Pós-decremento	Unário	1
<code>++</code>	Pré-incremento	Unário	2
<code>--</code>	Pré-decremento	Unário	2
<code>*</code>	Multiplicação	2 operandos	13
<code>/</code>	Divisão	2 operandos	13
<code>+</code>	Adição	2 operandos	14
<code>%</code>	Módulo (resto da divisão)	2 operandos	14
<code>-</code>	Subtração	2 operandos	14

- **Incremental (`++`)**

- **Pré-incremental ou prefixo**

Significa que, se o sinal for colocado antes da variável, primeiramente será somado o valor **1** para essa variável, continuando em seguida a resolução da expressão.

- **Pós-incremental ou sufixo**

Significa que, se o sinal for colocado após a variável, primeiro será resolvida toda a expressão (adição, subtração, multiplicação, etc.) para, em seguida, ser adicionado o valor **1** à variável.

- **Decremental (`--`)**

- **Pré-decremental ou prefixo**

Significa que, se o sinal for colocado antes da variável, primeiramente será subtraído o valor **1** dessa variável e, em seguida, será dada continuidade à resolução da expressão.

- **Pós-decremental ou sufixo**

Significa que, se o sinal for colocado após a variável, primeiro será resolvida toda a expressão (adição, subtração, multiplicação etc.) para, em seguida, ser subtraído o valor 1 da variável.

- **Exemplo 1:**

```
int a = 3, b = 5, c = 10;
int d = ++a * b-- - c++;
label1.Text = string.Format("a = {0}, b = {1}, c = {2}, d = {3}",
                            a, b, c, d);
```

Resultado final:

a = 4, b = 4, c = 11, d = 10

- **Exemplo 2:**

```
int a = 3, b = 5, c = 10;
int d = c % a + b * a - c++ - ++a;
int e = c / b * 2;
label1.Text =
    string.Format("a = {0}, b = {1}, c = {2}, d = {3}, e = {4}",
                  a, b, c, d, e);
```

Resultado final:

a = 4, b = 5, c = 11, d = 2, e = 4

Devemos estar atentos para o fato de que nem todos os operadores podem ser utilizados com todos os tipos de dados. Embora existam valores que aceitam qualquer um dos operadores aritméticos (a saber: **char**, **decimal**, **double**, **float**, **int** e **long**), há também aqueles que não aceitam (como **bool** e **string**, nos quais só podemos aplicar o operador **+**).

Quanto ao resultado de uma operação aritmética, ele depende não só do operador, mas também do operando, ou seja, quando temos operandos de mesmo tipo, obtemos um resultado que também é desse tipo; quando temos operandos de tipos diferentes, o compilador do C#, detectando a incompatibilidade, gera um código que converte um dos valores antes que se execute a operação.

Embora o compilador do C# tenha a capacidade de fazer as devidas conversões antes de executar operações em que os operandos (ou valores) são de tipos diferentes, essa não é uma prática recomendada.

3.5.3. Operadores relacionais

Operadores relacionais (ou booleanos) são aqueles que realizam uma operação cujo resultado é sempre **true** ou **false**. Dentre os diversos operadores booleanos existentes, o mais simples é o **NOT**, representado por um ponto de exclamação (!), o qual nega um valor booleano, apresentando sempre como resultado um valor oposto.

Operador	Descrição	Tipo	Preced.
<	Menor que	2 operandos	16
>	Maior que	2 operandos	16
>=	Maior ou igual	2 operandos	16
<=	Menor ou igual	2 operandos	16
==	Igual	2 operandos	17
!=	Diferente	2 operandos	17
is	Testa se um objeto é de uma determinada classe.	2 operandos	17

- **Exemplo 1:**

```

int a = 3, b = 5, c = 10;
bool d = a > b * 2;
bool e = a * c <= b++ * --c;
bool f = a * b == --a * b;
label1.Text = string.Format(
    "a = {0}, b = {1}, c = {2}, d = {3}, e = {4} f = {5}",
    a, b, c, d, e, f);

```

Resultado Final

a = 2, b = 6, c = 9, d = False, e = True, f = False

- **Exemplo 2:**

```

int a = 3, b = 5, c = 10;
bool d = c > b / 2;
bool e = a + c-- >= --b * c++;
bool f = a * ++b < --a * c;
label1.Text = string.Format(
    "a = {0}, b = {1}, c = {2}, d = {3}, e = {4} f = {5}",
    a, b, c, d, e, f);

```

Resultado Final

a = 2, b = 5, c = 10, d = True, e = False, f = True

3.5.4. Operadores lógicos

Os operadores lógicos trabalham com operandos booleanos e seu resultado também será booleano (**true** ou **false**). Sua função é combinar duas expressões ou valores booleanos em um só. Utilizados somente em expressões lógicas, eles estão reunidos na seguinte tabela:

Operador	Descrição	Tipo	Precedência
!	NÃO lógico (NOT)	2 operandos	8
&&	E lógico (AND)	2 operandos	22
	OU lógico	2 operandos	23



Embora sejam parecidos, a diferença entre os operadores lógicos e os operadores relacionais e de igualdade é que não só o valor das expressões em que eles aparecem é verdadeiro ou falso, mas também os valores em que operam.

Em um teste lógico utilizando o operador **&&** (AND), o resultado somente será verdadeiro (**true**) se ambas as expressões lógicas forem avaliadas como verdadeiras. Porém, se o operador utilizado for **||** (OR), basta que uma das expressões lógicas seja verdadeira para que o resultado também seja verdadeiro. Já o operador lógico **!** (NOT) é utilizado para gerar uma negação, invertendo a lógica de uma expressão.

- **Exemplo:**

```
int a = 3, b = 5, c = 10;
bool d = a > b * 2 && b > c || 2 * a <= b;
bool e = a * 2 <= c && c <= 10 && c < 2 * b | true;
bool f = 2 * b == c || !(a + 2 != b) && b > a;
label1.Text = string.Format(
    "a = {0}, b = {1}, c = {2}, d = {3}, e = {4}, f = {5}",
    a, b, c, d, e, f);
```

Resultado Final

a = 3, b = 5, c = 10, d = False, e = True, f = True

Quando trabalhamos com os operadores **||** e **&&**, pode ser que eles não façam a avaliação dos dois operandos para que, então, determinem o resultado de uma expressão lógica. Isso ocorre devido a um recurso conhecido como **curto-circuito** que, em alguns casos, ao avaliar o primeiro operando da expressão, já pode determinar seu resultado. Esse resultado baseado em apenas um dos operandos é possível quando:

- o operador localizado à esquerda de **&&** for **false**, o que torna desnecessária a avaliação do segundo, uma vez que o resultado será, obrigatoriamente, **false**;
- o operador localizado à esquerda de **||** for **true**, o que torna desnecessária a avaliação do segundo, uma vez que o resultado será, obrigatoriamente, **true**.

Ao elaborarmos as expressões de modo que a adoção dessa prática se torne possível, ou seja, posicionando expressões booleanas simples à esquerda, o desempenho do código pode ser otimizado, já que muitas expressões complexas, se posicionadas à direita, não precisarão passar por avaliação alguma.

3.5.5. Operador ternário

O operador ternário, ou operador condicional, é composto por três operandos separados pelos sinais ? e : e tem o objetivo de atribuir um valor a uma variável de acordo com o resultado de um teste lógico. Sua sintaxe é a seguinte:

```
teste lógico ? valor se verdadeiro : valor se falso;
```

Em que:

- **teste lógico** é qualquer expressão ou valor que pode ser avaliado como verdadeiro ou falso;
- **valor se verdadeiro** é o valor atribuído se o teste lógico for avaliado como verdadeiro;
- **valor se falso** é o valor atribuído se o teste lógico for avaliado como falso.

O código a seguir ilustra a utilização do operador ternário:

```
// gerador de números aleatórios
Random rn = new Random();
// gera um número aleatório no intervalo de 0 a 99
int r = rn.Next(0, 100);
// testa se o número gerado é par
label1.Text = r.ToString() +
(r % 2 == 0 ? "É PAR" : "É ÍMPAR");
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Instrução é um comando responsável pela execução de uma determinada ação;
- Método é uma sequência nomeada de instruções;
- Sintaxe é o conjunto de instruções que obedecem a uma série de regras específicas para seu formato e construção;
- As variáveis podem ser definidas como locais destinados ao armazenamento temporário de informações de diferentes tipos, como números, palavras, propriedades e datas, entre outras;
- Os operadores indicam a operação matemática que será executada, gerando novos valores a partir de um ou mais operandos (itens à direita ou esquerda do operador).

3

Instruções, tipos de dados, variáveis e operadores

Teste seus conhecimentos

SOUZA
07-03
CARLOS
77.



IMPACTA
EDITORA

1. Qual o tipo de dado mais adequado para declarar uma variável que armazena o preço de um produto?

- a) int
- b) long
- c) byte
- d) decimal
- e) uint

2. Considerando as seguintes variáveis e a expressão lógica adiante, qual a resposta correta?

```
int a = 10, b = 15, c = 10;  
bool d = a == c || b != a && a == b;
```

- a) False
- b) !=
- c) True
- d) &*
- e) !<>

3. Considerando o código a seguir, qual será o resultado mostrado em label1?

```
int a = 4, b = 5, c = 6;
int d = (a++ / 2) * (++c % 2) * ++b;
string e = d + (d % 3 == 0 ? "é " : "não é ") + "múltiplo de 3";
label1.Text = string.Format(
    "a = {0}, b = {1}, c = {2}, d = {3}, e = {4}", a, b, c, d, e);
```

- a) a = 5, b = 6, c = 8, d = 11, 11 não é múltiplo de 3.
- b) a = 4, b = 5, c = 9, d = 15, 15 é múltiplo de 3.
- c) a = 5, b = 6, c = 7, d = 12, 12 não é múltiplo de 3.
- d) a = 5, b = 6, c = 7, d = 12, 12 é múltiplo de 3.
- e) a = 5, b = 6, c = 7, d = 15, 15 é múltiplo de 3.

4. Qual tipo de dado é mais adequado para armazenar a quantidade filhos que uma pessoa tem?

- a) int
- b) uint
- c) byte
- d) sbyte
- e) long

5. A que se refere a sintaxe a seguir?

```
teste lógico ? valor se verdadeiro : valor se falso;
```

- a) A um operador unário.
- b) A um operador binário.
- c) A um operador ternário.
- d) A um operador lógico reduzido.
- e) A nada.

3

Instruções, tipos de dados, variáveis e operadores

Mãos à obra!

SOUZA
CARLOS
2013

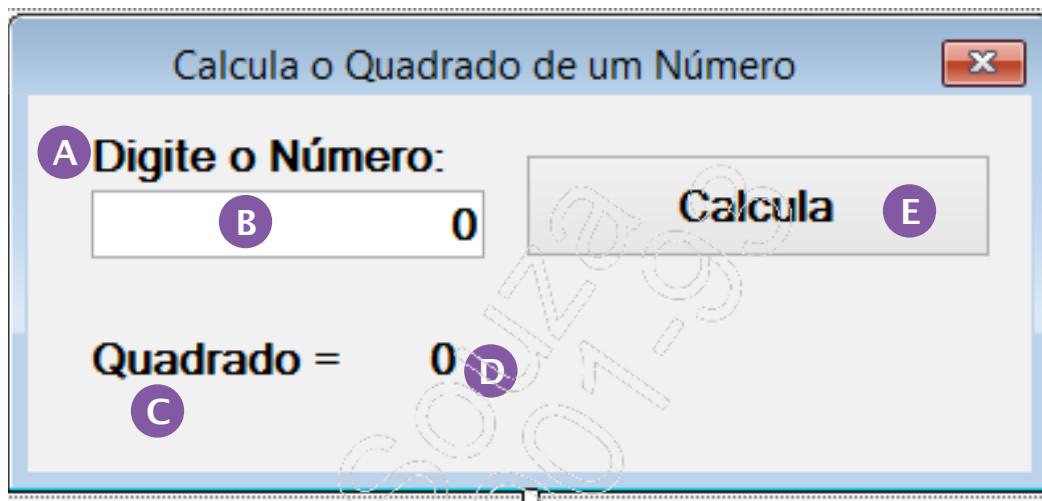


IMPACTA
EDITORA

Laboratório 1

A – Criando um projeto que calcula o quadrado de um número

1. Crie um novo projeto Windows Forms chamado **Quadrado** e monte a tela de acordo com a imagem a seguir:



	Controle	Propriedade	Conteúdo
A	Label	Text	Digite o Número:
		Name	numeroTextBox
B	TextBox	Text	0
		TextAlign	Right
C	Label	Text	Quadrado
D	Label	Name	quadradoLabel
		Text	0
E	Button	Name	calculaButton
		Text	Calcula

- **Propriedades do formulário:**

- **AcceptButton:** `calculaButton`. Faz com que o botão seja executado quando pressionarmos a tecla ENTER;
- **FormBorderStyle:** `FixedDialog`. Impede redimensionar o Form;
- **MaximizeBox:** `false`. Oculta o botão maximizar;
- **MinimizeBox:** `false`. Oculta o botão minimizar;
- **StartPosition:** `CenterScreen`. Centraliza o formulário no vídeo;
- **Text:** Calcula o quadrado de um número.

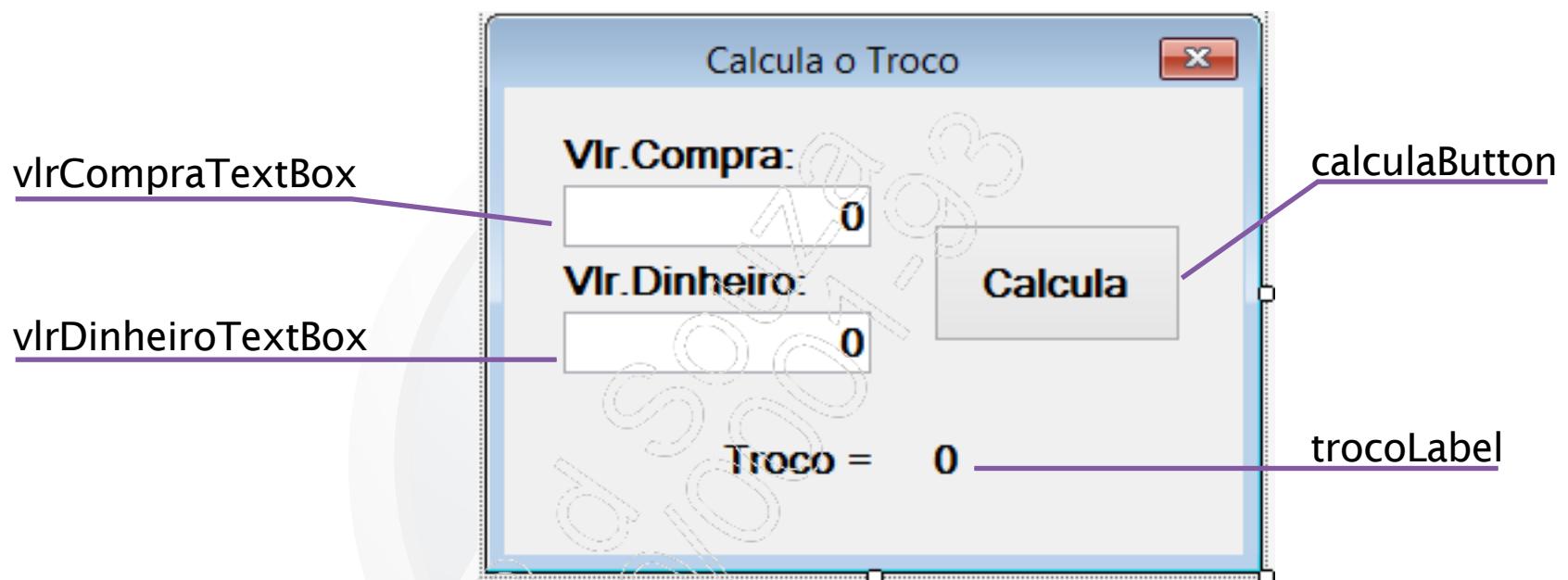
- **Evento Click do botão calculaButton:**

```
private void calculaButton_Click(object sender, EventArgs e)
{
    // declarar uma variável para cada dado envolvido no processo
    int numero, quadrado;
    // receber os dados de entrada
    numero = Convert.ToInt32(numeroTextBox.Text);
    // calcular os dados de saída
    quadrado = numero * numero;
    // mostrar o resultado na tela
    quadradoLabel.Text = quadrado.ToString();
}
```

Laboratório 2

A - Criando um projeto que calcula o troco de uma compra

1. Crie um novo projeto Windows Forms chamado **Troco** e monte a tela de acordo com a imagem a seguir. Os quadrinhos indicam a propriedade **Name** de cada controle. Aqueles que não tiverem o quadrinho podem ficar com o nome padrão.



- Complete o evento Click do botão **calculaButton**:

```
private void btnCalcula_Click(object sender, EventArgs e)
{
    // 1. declarar uma variável para cada dado envolvido no
    //    processo, neste caso 3 variáveis. Como estes dados
    //    podem ter casas depois da vírgula, declare as
    //    variáveis do tipo decimal.

    // 2. receber os dados de entrada. Converter para decimal
    //    a propriedade Text de cada TextBox e armazenar o resultado nas
    //    variáveis declaradas no passo número 1

    // 3. fazer o cálculo e armazenar o resultado na variável troco

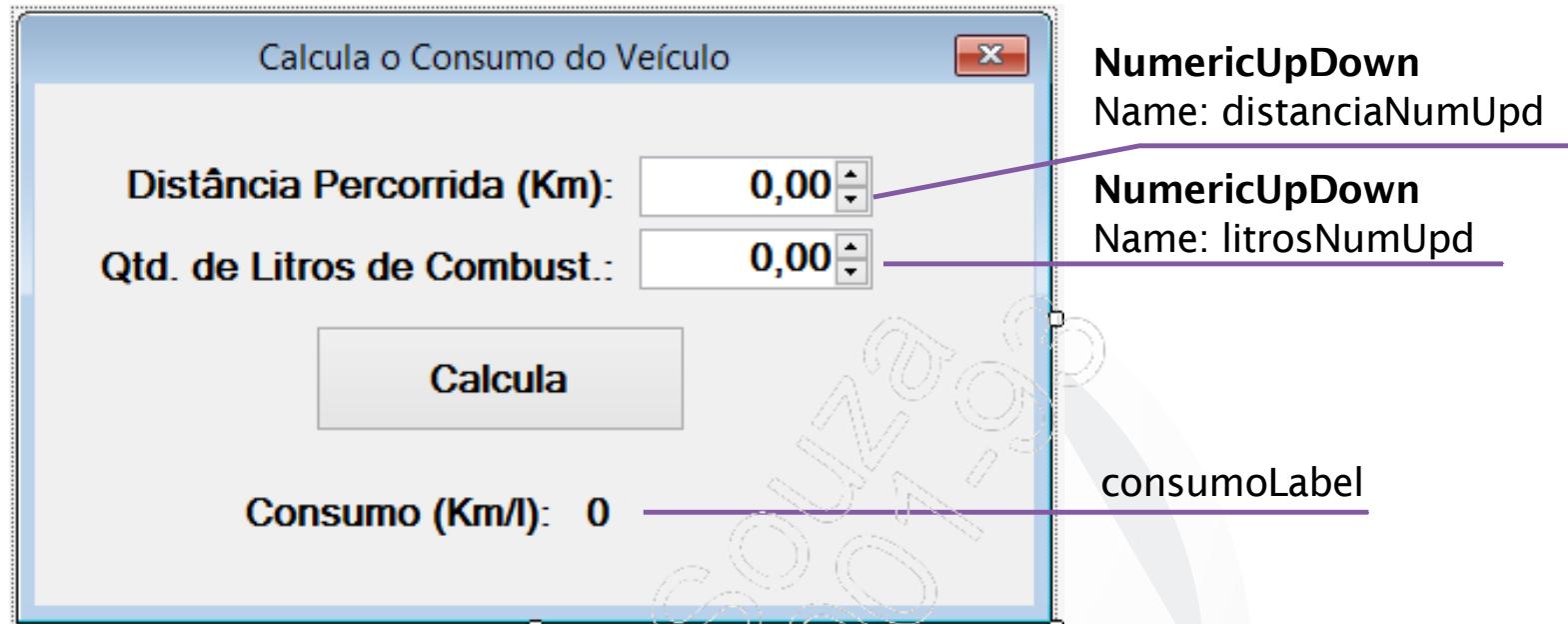
    // 4. exibir o resultado em trocoLabel

}
```

Laboratório 3

A – Criando um projeto para calcular o consumo de combustível de um veículo

- Crie um novo projeto chamado **Consumo** e monte a tela a seguir:



- Utilize o componente **NumericUpDown** para receber os dados numéricos. Este componente possui uma propriedade chamada **Value** que já armazena o número digitado no tipo Decimal. Para os dois componentes **NumericUpDown**, altere as propriedades:

- TextAlign: Right;**
- DecimalPlaces: 2;**
- Maximum: 1000.**

- Confira o evento **Click** do botão **calculaButton**:

```

private void btnCalcula_Click(object sender, EventArgs e)
{
    // 1. Declarar as variáveis de entrada (distância e litros)
    //     e as variáveis de resultado (consumo)

    // 2. armazenar os conteúdos dos NumericUpDown nas variáveis de
    //     entrada

    // 3. fazer o cálculo (consumo = distância / litros)

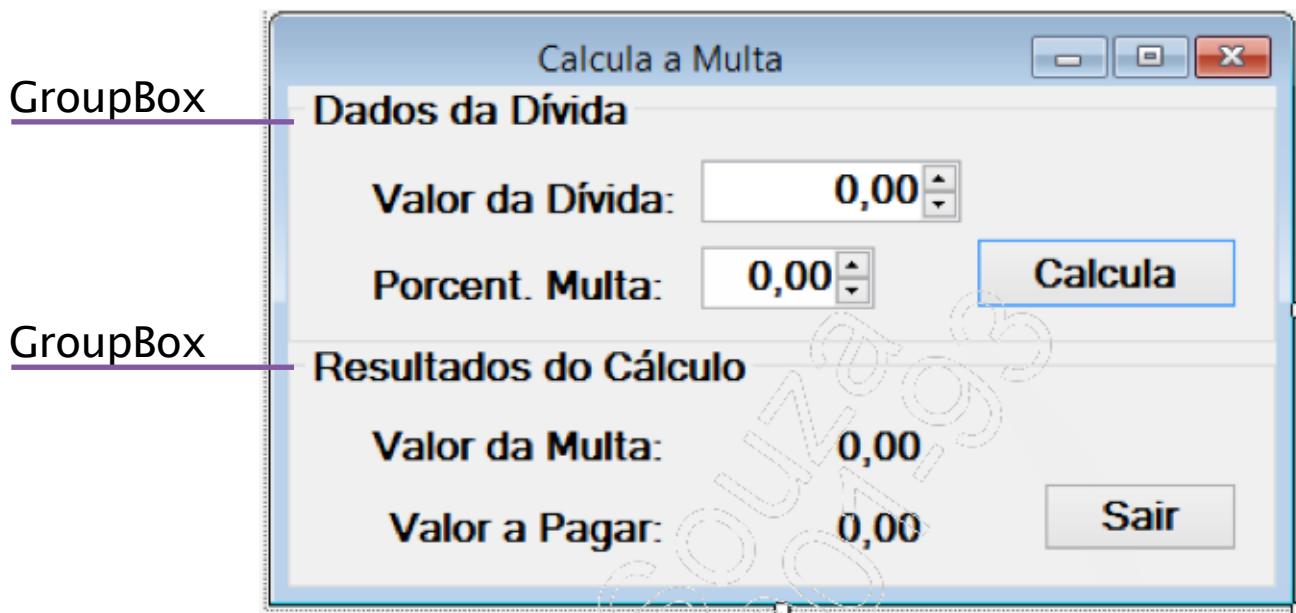
    // 4. mostrar o resultado na tela
}

```

Laboratório 4

A - Criando um projeto para calcular o valor de uma multa

1. Crie um novo projeto chamado **Multa** e monte a tela a seguir:



- Na área de entrada:
 - **NumericUpDown**: dividaNumUpd;
 - **NumericUpDown**: porcNumUpd;
 - **Button**: calculaButton.
- Na área de resultados:
 - **Label**: vlrMultalabel;
 - **Label**: vlrPagarlabel;
 - **Button**: sairbutton.
- **Cálculos:**
 - **vlrMult = divida * porc / 100;**
 - **vlrPagar = divida + vlrMult.**

2. Faça o evento **Click** do botão **calculaButton**;
3. Faça o evento **Click** do botão **sairButton**: **this.Close()**;

Instruções de decisão

4

- ✓ Instruções de decisão if / else e switch / case.

Carlos M. H. Souza
77.357.98000-007-03



IMPACTA
EDITORA

4.1. Introdução

Neste capítulo, veremos de que formas é possível instruir o C# a tomar determinadas decisões em alguns pontos específicos do programa. Para que esta ou aquela ação seja realizada, o programa deve avaliar expressões e, a partir dessa avaliação, especificamos o que deve ser feito.

4.2. Instruções de decisão if / else e switch / case

Para decidirmos qual ação deverá ser tomada em determinada parte de um programa, solicitamos que este avalie uma expressão. Caso essa expressão seja avaliada como verdadeira, uma sequência de comandos será executada.

A seguir, serão abordadas as instruções de decisão **if / else** e **switch / case**.

4.2.1. If / else

Quando utilizamos uma instrução **if**, apenas expressões com resultados booleanos podem ser avaliadas. Para definir uma instrução **if**, deve-se empregar a seguinte sintaxe:

```
if (testeCondisional)
{
    comandosTrue;
}
else // esta parte é opcional
{
    comandosFalse
}
```

O **testeCondisional** precisa estar entre parênteses e pode ser qualquer expressão que retorne **true** ou **false**. Em caso verdadeiro, o bloco de instruções que se encontra entre as chaves (**{ }{ }**) será executado. Se o resultado for falso, o bloco de comandos não será executado, e o processamento prossegue após o bloco de comandos. Se utilizarmos o **else**, ele será executado quando a condição for falsa e antes de prosseguir após o **if**.

As chaves ({}) do bloco verdadeiro ou do falso poderão ser omitidas caso estes blocos possuam apenas uma única instrução.

A maneira como empregamos a instrução if em conjunto com a instrução else no C# é bastante similar ao seu uso em outras linguagens de programação.

Vejamos agora alguns exemplos com a instrução **if / else**:

- Reabra o projeto chamado **Troco**, criado no laboratório do capítulo anterior. Vamos alterar o método que faz o cálculo incluindo um IF para verificar a validade dos dados de entrada, ou seja, se o valor dado em pagamento for menor que o valor da compra, forneceremos uma mensagem;

```
private void btnCalcula_Click(object sender, EventArgs e)
{
    // declarar uma variável para cada dado
    decimal vlrCompra, vlrPago, troco;
    // receber os dados de entrada
    vlrCompra = Convert.ToDecimal(vlrCompraTextBox.Text);
    vlrPago = Convert.ToDecimal(vlrDinheiroTextBox.Text);
```

```
        if (vlrPago < vlrCompra)
        {
            // abre janela de mensagem
            MessageBox.Show("Valor pago não é suficiente...");
```

```
            // finaliza a execução deste método
            return;
        }

        // fazer os cálculos
        troco = vlrPago - vlrCompra;
        // exibir o resultado
        trocoLabel.Text = troco.ToString("#,##0.00");
    }
```

C# - Módulo I

- Reabra o projeto chamado **Multa**, também criado no laboratório do capítulo anterior. Vamos alterar o método que faz o cálculo incluindo um IF para verificar a validade dos dados de entrada, ou seja, os valores digitados (valor da dívida e porcentagem de multa) precisam ser maiores que zero:

```
private void calculaButton_Click(object sender, EventArgs e)
{
    // variáveis de entrada
    decimal divida, porcMultta;
    // variáveis de saída (resultados)
    decimal multa, vlrPagar;
    // receber os dados digitados na tela
    divida = dividaNumUpd.Value;
    porcMultta = porcNumUpd.Value;

    // se o valor da dívida OU a porc. de multa for zero ou negativo
    if (divida <= 0 || porcMultta <= 0)
    {
        // abrir janela de mensagem
        MessageBox.Show("Os valores devem ser positivos");
        // finalizar a execução deste método
        return;
    }

    // fazer os cálculos
    multa = divida * porcMultta / 100;
    vlrPagar = divida + multa;
    // mostrar os resultados nos labels
    vlrMulttaLabel.Text = multa.ToString("#,##0.00");
    vlrPagarLabel.Text = vlrPagar.ToString("#,##0.00");
}
```

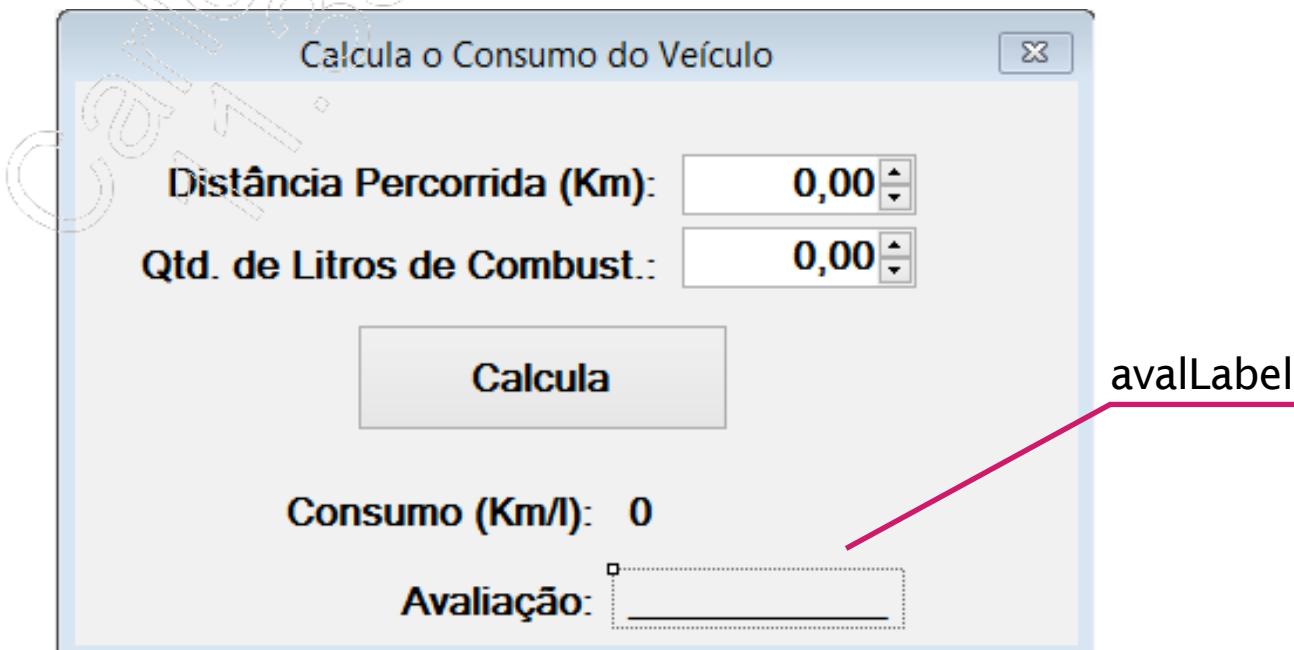
- Reabra o projeto chamado **Consumo**, criado no laboratório do capítulo anterior. A quantidade de litros não pode ser zero, pois não existe divisão por zero. E também não pode ser negativa, porque não há como colocar uma quantidade de litros negativa no tanque:

```

private void btnCalcula_Click(object sender, EventArgs e)
{
    // declarar as variáveis de entrada (distancia e litros)
    // e as variáveis de resultado (consumo)
    decimal distancia, litros, consumo;
    // converter os dados digitados na tela e colocá-los nas
    // variáveis de entrada
    distancia = distanciaNumUpd.Value;
    litros = litrosNumUpd.Value;
    if (litros <= 0)
    {
        MessageBox.Show("A quantidade de litros deve ser positiva");
        return;
    }
    // fazer os cálculos
    consumo = distancia / litros;
    // mostrar o resultado na tela
    consumoLabel.Text = consumo.ToString("0.00");
}

```

- Ainda no projeto chamado **Consumo**, altere o formulário incluindo os controles mostrados na imagem a seguir:



C# - Módulo I

Quando clicarmos no botão, o label **avalLabel** deve exibir uma das mensagens a seguir:

Consumo	Mensagem
Até 4 Km/l	Péssimo
Até 6 Km/l	Ruim
Até 8 Km/l	Regular
Até 12 Km/l	Bom
Acima de 12 Km/l	Ótimo

Existem duas soluções, mas a segunda é melhor porque as condições são mutuamente exclusivas, ou seja, é impossível ter duas situações simultaneamente verdadeiras.

```
// fazer os cálculos
consumo = distância / litros;
// mostrar o resultado na tela
consumoLabel.Text = consumo.ToString("0.00");

// SOLUÇÃO 1 - Não é boa, pois avalia todos os IFs
if (consumo <= 4) avalLabel.Text = "Péssimo";
if (consumo > 4 && consumo <= 6) avalLabel.Text = "Ruim";
if (consumo > 6 && consumo <= 8) avalLabel.Text = "Regular";
if (consumo > 8 && consumo <= 12) avalLabel.Text = "Bom";
if (consumo > 12) avalLabel.Text = "Ótimo";

// SOLUÇÃO 2 - Boa, pois encontrando um IF verdadeiro
// não avalia os outros
if (consumo <= 4) avalLabel.Text = "Péssimo";
else if (consumo <= 6) avalLabel.Text = "Ruim";
else if (consumo <= 8) avalLabel.Text = "Regular";
else if (consumo <= 12) avalLabel.Text = "Bom";
else avalLabel.Text = "Ótimo";
```

4.2.2. Switch / case

Quando temos diversas instruções **if** em cascata avaliando uma expressão idêntica, podemos usar **switch / case**, desde que sejam atendidas as seguintes exigências:

- A expressão avaliada pelo switch tem que ser de um tipo INTEIRO, BOOL, STRING ou ENUMERADO;
- O resultado avaliado tem que ser exato, ou seja, não podemos usar maior, menor, &&, ||, etc.

Vejamos a sintaxe de **switch**:

```
switch (expressão)
{
    case valor1:
        instruções a executar se a expressão for igual a valor1
        break;
    case valor2:
        instruções a executar se a expressão for igual a valor2
        break;
    case valor3:
        instruções a executar se a expressão for igual a valor3
        break;
    ...
    default
        instruções a executar se a expressão não for igual a valor
        algum
        break;
}
```

O funcionamento da sintaxe de **switch** pode ser descrita da seguinte forma:

1. A **expressão** é avaliada;
2. O controle passa para o bloco de código em que **valor** é igual à **expressão**;
3. A execução do código é realizada até que se atinja a instrução **break**;
4. A instrução **switch** é finalizada;

5. O programa prossegue em sua execução a partir da instrução localizada após a chave de fechamento de **switch**.

Apenas nos casos em que não for encontrado um valor igual à expressão, as instruções do rótulo **default** serão executadas. Se tal rótulo não tiver sido criado, a execução continuará a partir da instrução imediatamente após a chave de fechamento de **switch**.

A seguir, temos um exemplo do uso de **switch / case**:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    string diaSemana = exemploTextBox.Text;
    string resultado = "";
    //Testa se existe algum valor para a variável diaSemana
    if (diaSemana.Trim() == "") { return; }
    switch (diaSemana)
    {
        case "1":
            resultado = "Segunda-Feira";
            break;
        case "2":
            resultado = "Terça-Feira";
            break;
        case "3":
            resultado = "Quarta-Feira";
            break;
        case "4":
            resultado = "Quinta-Feira";
            break;
        case "5":
            resultado = "Sexta-Feira";
            break;
        case "6":
            resultado = "Sábado";
            break;
        case "7":
            resultado = "Domingo";
            break;
        default:
            resultado = "Informe um dia de semana entre 1 e 7";
            break;
    }
    exemploListBox.Items.Add(resultado);
    exemploListBox.Items.Add(new String('-', 90));
}
```

Para executar as mesmas instruções para mais de um valor, deve-se inserir uma lista de rótulos **case** com as instruções localizadas apenas ao final da lista. Se para algum rótulo existir uma ou mais instruções associadas, será gerado um erro pelo compilador, impedindo que a execução prossiga para os próximos rótulos. Portanto, quando uma opção **case** possui o mesmo resultado de uma outra opção **case**, podemos escrever o código assim:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    string diaSemana = exemploTextBox.Text;
    string resultado = "";
    //Testa se existe algum valor para a variável diaSemana
    if (diaSemana.Trim() == "") { return; }
    switch (diaSemana)
    {
        case "1":           //Caso diaSemana seja igual a uma
        case "2":           //dessas opções, a instrução
        case "3":           //antes do BREAK será executada.
        case "4":           //Caso seja diferente, o
        case "5":           //próximo CASE será verificado.
            resultado = "Dia útil";
            break;
        case "6":
        case "7":
            resultado = "Final de semana";
            break;
        default:
            resultado = "Informe um dia de semana entre 1 e 7";
            break;
    }
    exemploListBox.Items.Add(resultado);
    exemploListBox.Items.Add(new string('-', 90));
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Para decidirmos qual ação deverá ser tomada em determinada parte de um programa, solicitamos que este avalie uma expressão. Caso essa expressão seja avaliada como verdadeira, uma sequência de comandos será executada;
- Quando utilizamos uma instrução **if**, apenas expressões com resultados booleanos podem ser avaliadas;
- Quando trabalhamos com a instrução **if** e desejamos executar comandos nos casos em que a condição é avaliada como falsa, é necessário digitar a cláusula **else** após as instruções da condição verdadeira e, na linha abaixo, inserir o bloco de instruções a serem executadas;
- A instrução **switch** é um modo de simular a utilização de várias instruções **if** e pode somente verificar uma relação de igualdade. É indicada para avaliar uma única expressão para mais de um valor possível, deixando o programa mais legível e tornando sua execução mais eficiente.

4

Instruções de decisão

Teste seus conhecimentos

Carlos M. 77.357.900-0003
SOUZA



IMPACTA
EDITORA

1. Assinale a alternativa que completa o código de modo que entre no if somente quando o funcionário ganhar menos que 1000 e tiver código de cargo igual a 3:

```
decimal salarioFuncionario = salarioNumericUpDown.Value;
decimal codDepFuncionario = codDepNumericUpDown.Value;
if (_____)
{
    salarioFuncionario *= 1.1;
}
```

- a) salarioFuncionario > 1000 && codDepFuncionario == 3
- b) salarioFuncionario < 1000 && codDepFuncionario = 3
- c) salarioFuncionario < 1000 || codDepFuncionario = 3
- d) salarioFuncionario < 1000 && codDepFuncionario == 3
- e) salarioFuncionario < 1000 || codDepFuncionario == 3

2. Assinale a alternativa que completa o código de modo que entre no if somente quando o funcionário ganhar menos que 1000 ou tiver código de cargo igual a 3:

```
decimal salarioFuncionario = salarioNumericUpDown.Value;
decimal codDepFuncionario = codDepNumericUpDown.Value;
if (_____)
{
    salarioFuncionario *= 1.1;
}
```

- a) salarioFuncionario > 1000 && codDepFuncionario == 3
- b) salarioFuncionario < 1000 && codDepFuncionario = 3
- c) salarioFuncionario < 1000 || codDepFuncionario = 3
- d) salarioFuncionario < 1000 && codDepFuncionario == 3
- e) salarioFuncionario < 1000 || codDepFuncionario == 3

3. Assinale a alternativa que completa o código de modo que entre no if somente quando o funcionário ganhar entre 1000 e 3000 (inclusive 1000 e 3000) e também tiver código de cargo igual a 3:

```
decimal salarioFuncionario = salarioNumericUpDown.Value;
decimal codDepFuncionario = codDepNumericUpDown.Value;
if (_____)
{
    salarioFuncionario *= 1.1;
}
```

- a) salarioFuncionario > 1000 && salarioFuncionario < 3000 && codDepFuncionario = 3
- b) salarioFuncionario >= 1000 || salarioFuncionario <= 3000 || codDepFuncionario == 3
- c) salarioFuncionario between 1000 && 3000 && codDepFuncionario == 3
- d) salarioFuncionario >= 1000 && salarioFuncionario <= 3000 && codDepFuncionario == 3
- e) salarioFuncionario >= 1000 || salarioFuncionario <= 3000 || codDepFuncionario == 3

4. Assinale a alternativa que completa o código de modo que entre no if somente quando o funcionário ganhar abaixo de 1000 ou acima de 3000 e, em ambos os casos, também tiver código do cargo igual a 3:

```
decimal salarioFuncionario = salarioNumericUpDown.Value;
decimal codDepFuncionario = codDepNumericUpDown.Value;
if (_____)
{
    salarioFuncionario *= 1.1;
}
```

- a) salarioFuncionario < 1000 && salarioFuncionario > 3000 && codDepFuncionario == 3
- b) salarioFuncionario < 1000 || salarioFuncionario > 3000 && codDepFuncionario == 3
- c) (salarioFuncionario < 1000 || salarioFuncionario > 3000) && codDepFuncionario == 3
- d) salarioFuncionario not between 1000 && 3000 && codDepFuncionario == 3
- e) salarioFuncionario ! between 1000 && 3000 && codDepFuncionario == 3

5. No código a seguir, se o conteúdo da variável `salarioFuncionario` for 3000, que valor será mostrado no Label no final do código?

```
decimal salarioFuncionario = salarioNumericUpDown.Value;
if (salarioFuncionario <= 1000) salarioFuncionario *= 1.5;
else if (salarioFuncionario <= 2000) salarioFuncionario *= 1.4;
else if (salarioFuncionario <= 5000) salarioFuncionario *= 1.2;
else salarioFuncionario *= 1.1;
novoSalarioLabel.Text = salarioFuncionario.ToString();
```

- a) 4500
- b) 4200
- c) 3600
- d) 3300
- e) O código provoca erro de compilação.

4

Instruções de decisão

Mãos à obra!

Carlos M. Souza
77.359.800-007-93

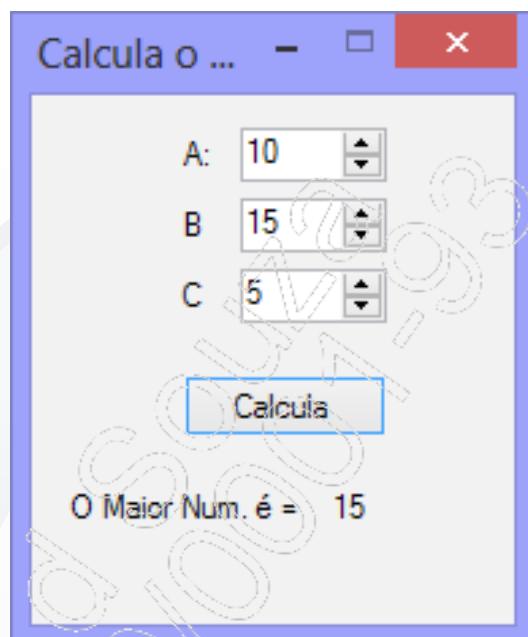


IMPACTA
EDITORA

Laboratório 1

A - Trabalhando com o projeto Maior

1. Abra o projeto **Maior** disponível na pasta **Cap_04\4_Maior_Lab1** fornecida pelo instrutor;



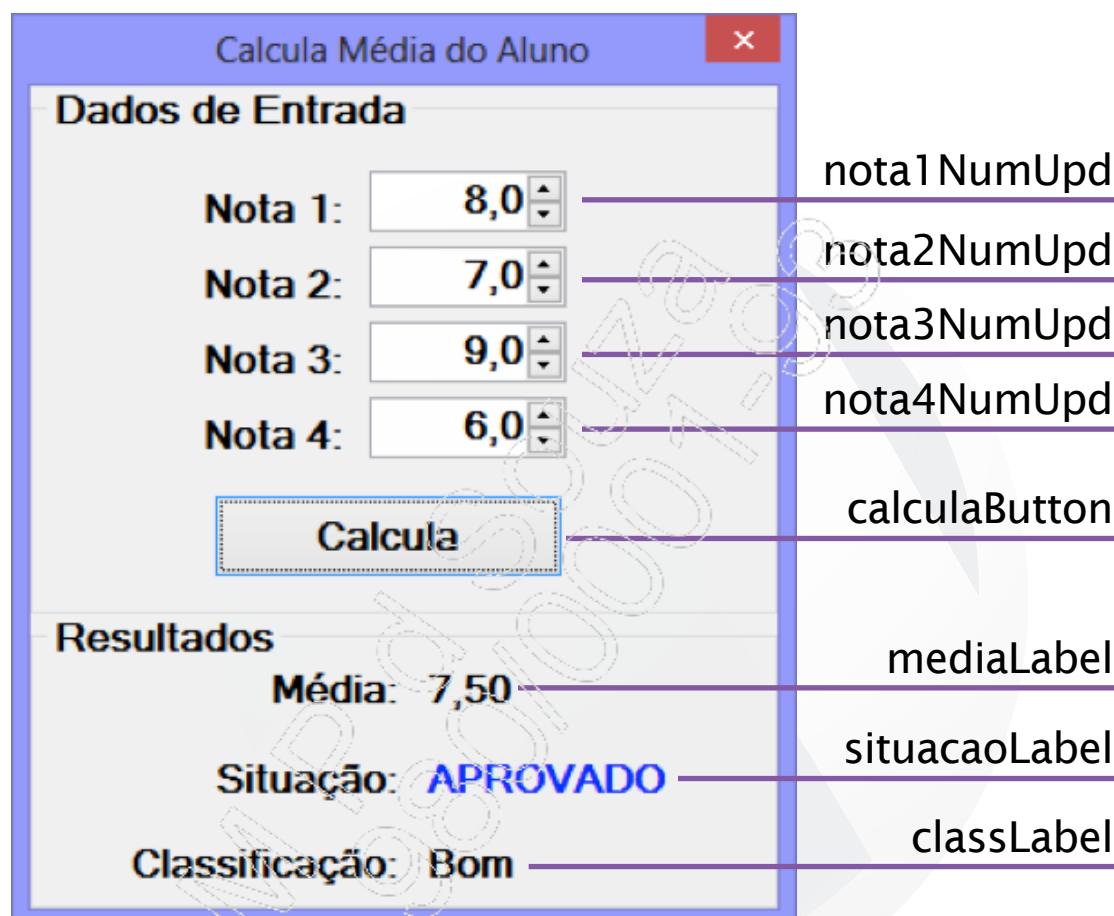
2. Complete o código do botão **Calcula** de modo que a variável **maior** armazene o maior dos três números.

```
private void btnCalcula_Click(object sender, EventArgs e)
{
    decimal a, b, c, maior;
    a = aNumUpd.Value;
    b = bNumUpd.Value;
    c = cNumUpd.Value;
    //-----COMPLETE-----  
  
    //-----  
    lblMaior.Text = maior.ToString();
}
```

Laboratório 2

A - Criando um projeto que calcula a média das notas dos alunos

1. Crie um novo projeto chamado **Media** e monte a tela a seguir:



1. Configure as propriedades **Minimum** e **Maximum** dos **NumericUpDown** para **0** e **10** respectivamente;

2. Crie um método para o evento **Click** do botão. Este método deverá realizar as seguintes ações:

- Calcular a média das quatro notas e exibi-la em **mediaLabel**;
- Se a média for menor que 5, o método deverá exibir, em **situacaoLabel**, a palavra **REPROVADO** na cor vermelha. Caso contrário, deverá imprimir a palavra **APROVADO** na cor azul;

- Deve exibir, em **classLabel**, a classificação do aluno de acordo com a tabela a seguir:

Média	Classificação
Abaixo ou igual a 2	PÉSSIMO
Acima de dois até 4	RUIM
Acima de 4 até 6	REGULAR
Acima de 6 até 8	BOM
Acima de 8 até 10	ÓTIMO

3. Toda vez que um **NumericUpDown** receber foco, todo o seu conteúdo deve ser selecionado. Para isso, siga estas instruções:

- Selecione os quatro componentes **NumericUpDown** e crie um método para o evento **Enter** (ganho de foco);
- Para descobrir qual dos quatro **NumericUpDown** ganhou foco, utilize:

```
NumericUpDown upd = (NumericUpDown)sender;
```

- Utilize o método **Select** da classe **Control** para selecionar todo o conteúdo do **NumericUpDown**.

```
void UpDownBase.Select(int start, int length)
```

Selects a range of text in the spin box (also known as an up-down control) specifying the starting position and number of characters to select.

start: The position of the first character to be selected.

length: The total number of characters to be selected.

Este é um help que o próprio editor de textos do Visual Studio exibe quando usamos algum comando da linguagem. Ele indica que o método **Select()** seleciona uma faixa de caracteres do texto contido no **NumericUpDown**. Devemos indicar a posição inicial da seleção e a quantidade de caracteres que queremos selecionar:

- **start:** A posição do primeiro caractere que será selecionado (zero corresponde ao primeiro);
- **length:** A quantidade de caracteres que serão selecionados.

Por exemplo, o código `textBox1.Select(3,5);` produz o seguinte efeito:



Instruções de repetição

5

- ✓ Instruções de repetição ou iteração.

Carloso M Pd SOUZA
77.357.980/0007-03



IMPACTA
EDITORA

5.1. Introdução

Neste capítulo, veremos como colocar um trecho de código em repetição ou loop.

5.2. Instruções de repetição ou iteração

Muitas vezes, precisamos repetir a execução de um bloco de códigos do programa até que uma determinada condição seja verdadeira, ou até atingir uma quantidade de execuções determinada. Para que essa repetição seja possível, utilizamos os laços de repetição de linguagem que, em C#, são os seguintes:

- **while**;
- **do / while**;
- **for**.

Nos subtópicos seguintes, veremos cada um deles mais detalhadamente e também estudaremos outros comandos utilizados em conjunto com as instruções **for**, **while** e **do / while**.

5.2.1. While

Quando não sabemos quantas vezes um determinado bloco de instruções precisa ser repetido, utilizamos o laço de repetição **while**. Com ele, a execução das instruções continuará enquanto uma determinada condição for verdadeira (true). A condição a ser analisada para execução do laço de repetição deverá retornar um valor booleano. Vejamos a sintaxe do laço de repetição **while**:

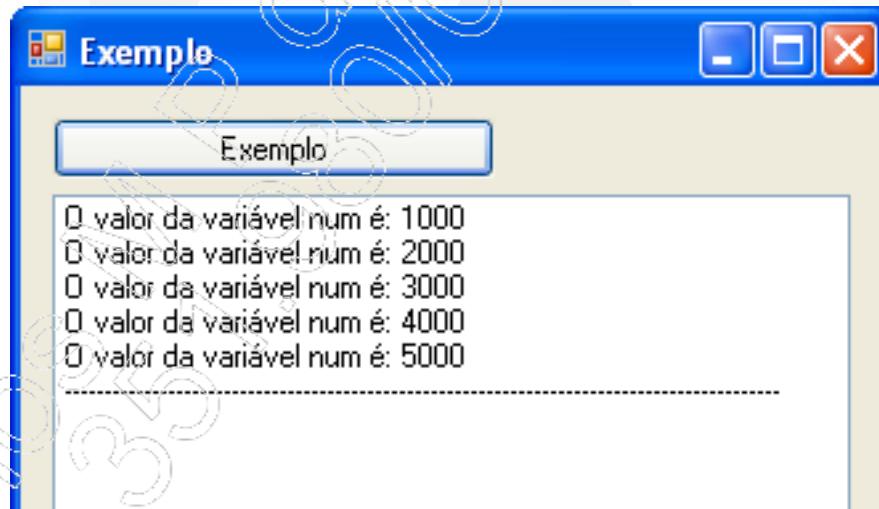
```
while (teste condicional)
{
    comandos;
}
```

No exemplo a seguir, o corpo do laço de repetição será executado somente se a condição for verdadeira. Assim, a execução se repetirá até que essa condição seja falsa. Vejamos:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    int num = 1000;

    //Enquanto num for menor ou igual a 5000
    while (num <= 5000)
    {
        exemploListBox.Items.Add(
            "O valor da variável num é: " + num);
        num += 1000;
    }
    exemploListBox.Items.Add(new string(' ', 90));
}
```

O resultado é o seguinte:



É importante saber que o laço de repetição **while** não será executado quando, na primeira verificação da condição, esta for falsa. Neste caso, o programa passará para a execução da próxima instrução após o laço.

! É fundamental incluir sempre uma expressão booleana que seja avaliada como **false**. Assim, definimos o ponto em que deve ser finalizado o loop e, consequentemente, evitamos a execução contínua do programa.

5.2.2. Do / while

A estrutura **do / while** tem basicamente o mesmo funcionamento da instrução **while**, porém, executa todos os comandos ao menos uma única vez, mesmo quando a condição não é verdadeira. Vejamos sua sintaxe:

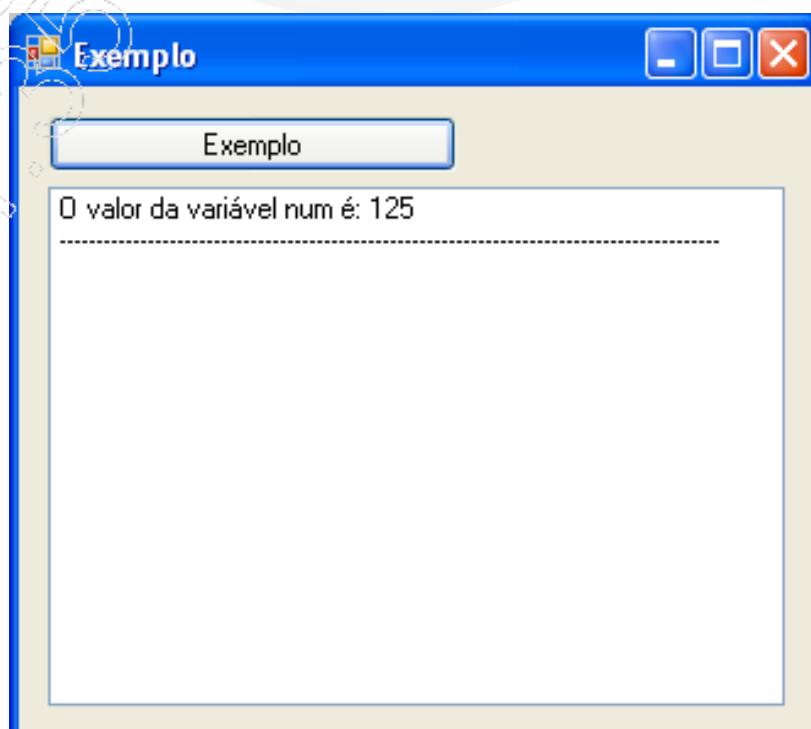
```
do
{
    comandos;
}
while(condição);
```

Observemos o exemplo a seguir:

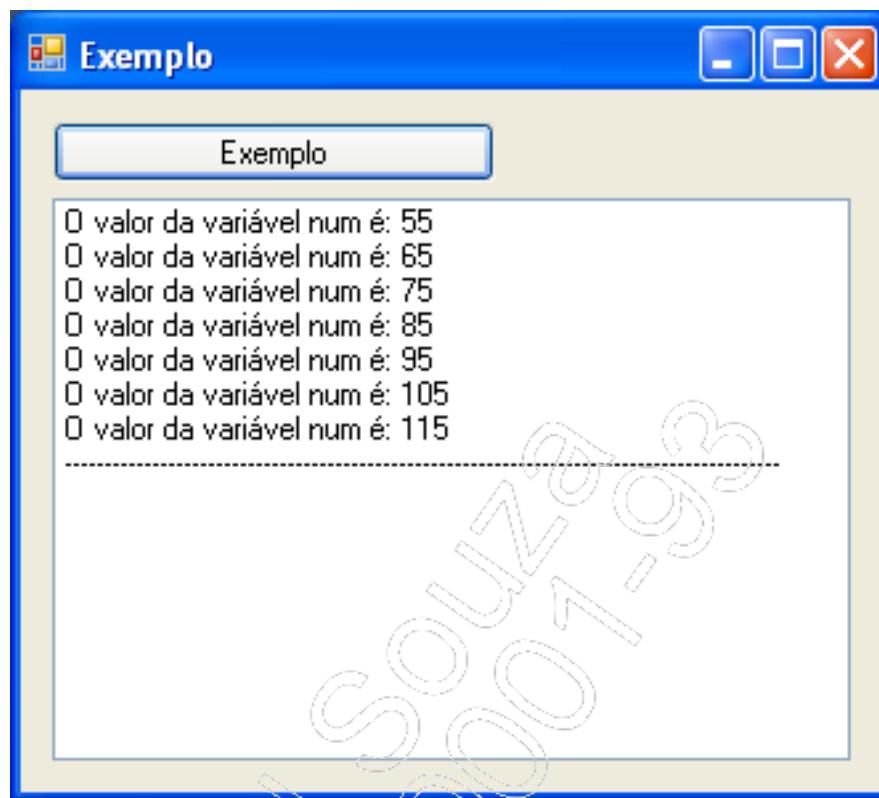
```
private void exemploButton_Click(object sender, EventArgs e)
{
    int num = 125;

    //Faça
    do
    {
        exemploListBox.Items.Add(
            "O valor da variável num é: " + num);
        num += 10;
    } while (num <= 120);
    exemploListBox.Items.Add(new string('-', 90));
}
```

O resultado é o seguinte:



As linhas de comando abaixo da instrução **do** foram executadas uma vez, já que a variável possui o valor **125** e, por isso, não atende à condição **while**. Ao alterarmos o valor dessa variável para **55**, por exemplo, obtemos o seguinte resultado:



5.2.3. For

A instrução **for** é usada para fazer um loop que inicia um contador em um valor específico. Ao executar o código que está dentro do loop, o contador é incrementado e, enquanto o limite estabelecido para o contador não for alcançado, o loop será repetido. Podemos utilizar o laço de repetição **for** quando soubermos exatamente quantas vezes um bloco de instruções deve ser repetido.

A declaração simples e mais comum do laço **for** é a seguinte:

```
for(declaração e inicialização de variáveis; expressão condicional; expressão de iteração)
{ }
```

Além do corpo do laço, **for** possui as seguintes partes principais: declaração e inicialização de variáveis, expressão condicional e expressão de iteração. A seguir, temos a descrição de cada uma dessas partes:

- **Declaração e inicialização de variáveis**

Nessa primeira parte da instrução **for**, podemos declarar e inicializar uma ou mais variáveis, quando houver. Elas são colocadas entre parênteses, após a palavra-chave **for** e, se houver mais de uma variável do mesmo tipo, elas são separadas por vírgulas, conforme o seguinte exemplo:

```
for (int a = 1, b = 1;
```

É importante lembrarmos que a declaração e a inicialização das variáveis, na instrução **for**, acontecem sempre antes de outros comandos. Elas ocorrem apenas uma vez no laço de repetição, sendo que o teste booleano e a expressão de iteração são executados a cada laço do programa.

- **Expressão condicional**

Na segunda parte da instrução do laço de repetição **for**, temos a expressão condicional. Ela se refere a um teste que será executado e deverá retornar um valor booleano. Por esse motivo, somente é válido especificarmos nessa parte da instrução **for** uma expressão lógica.

Devemos observar que a expressão lógica pode ser complexa e, também, ter cuidado com os códigos que utilizam diversas expressões lógicas. Vejamos no exemplo seguinte uma expressão condicional válida:

```
for (int a = 1, b = 1; (a < 300 && b < 300);
```

A seguir, temos um exemplo de expressão condicional não válida:

```
for (int a = 1, b = 1; (a < 300), (b < 300));
```

Nesse exemplo, há dois testes booleanos separados por vírgulas, o que faz com que a instrução não seja executada e gere um erro. Isso ocorre porque podemos ter apenas uma expressão de teste na sintaxe desse comando.

- **Expressão de iteração**

Na expressão de iteração, indicamos o que deverá ocorrer após cada execução do corpo do laço de repetição. Ela sempre será processada após o corpo do laço ser executado, mas será a última execução da instrução do laço **for**. Na linha seguinte, temos a sintaxe completa da instrução **for**:

```
for(declaração e inicialização de variáveis; expressão condicional;  
     expressão de iteração)  
{  
    instrução do corpo do laço for;  
}
```

Vejamos o exemplo a seguir:

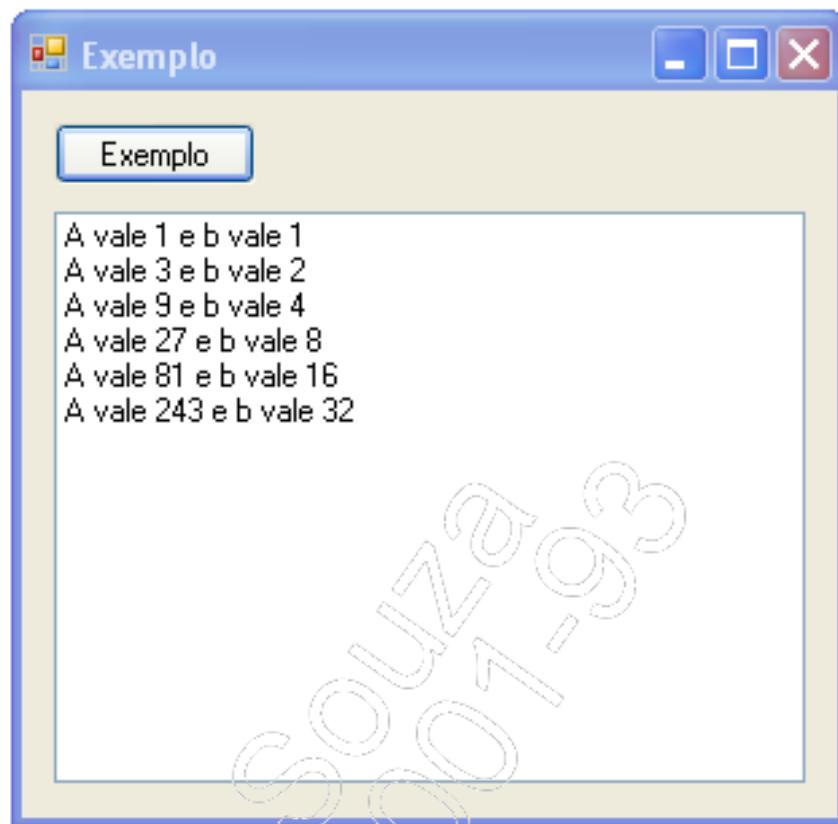
```
for (int a = 1, b = 1; (a < 300 && b < 300);  
    exemploListBox.Items.Add("a vale " + a + " e b vale " + b),  
    a *= 3, b *= 2)  
{ }
```

Em que:

- **int** refere-se ao tipo de variável que está sendo declarada;
- **a = 1** e **b = 1** referem-se às variáveis que estão sendo inicializadas;
- **(a < 300 && b < 300)** são testes booleanos que serão feitos nas variáveis;
- **a *= 3, b *= 2** são as iterações das variáveis **a** e **b**.

C# - Módulo I

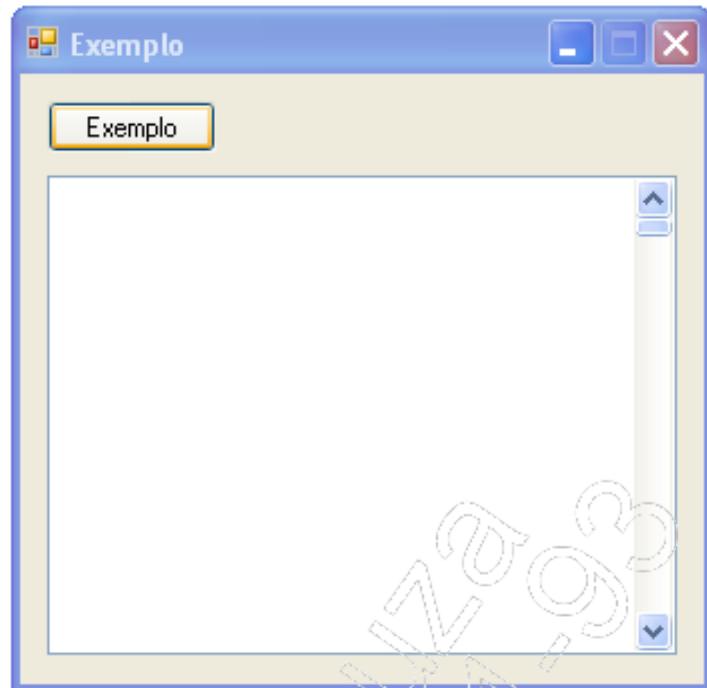
Vejamos o resultado da execução desse código:



Após verificarmos todas essas informações, podemos deixar de declarar uma das três partes descritas do laço de repetição **for**, apesar de essa não ser uma prática recomendada. Nesse caso, o laço será infinito e, não havendo seções de declaração e inicialização no laço de repetição **for**, ele agirá como um laço de repetição **while**. Observemos:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    for (; ; )
    {
        exemploListBox.Items.Add(
            "Laço for sem declaração e inicialização de variáveis");
    }
}
```

O resultado da execução desse laço infinito será o seguinte:



Em relação ao escopo das variáveis que foram declaradas no laço de repetição **for**, o mesmo é finalizado juntamente com o laço. Sendo assim, devemos observar o seguinte:

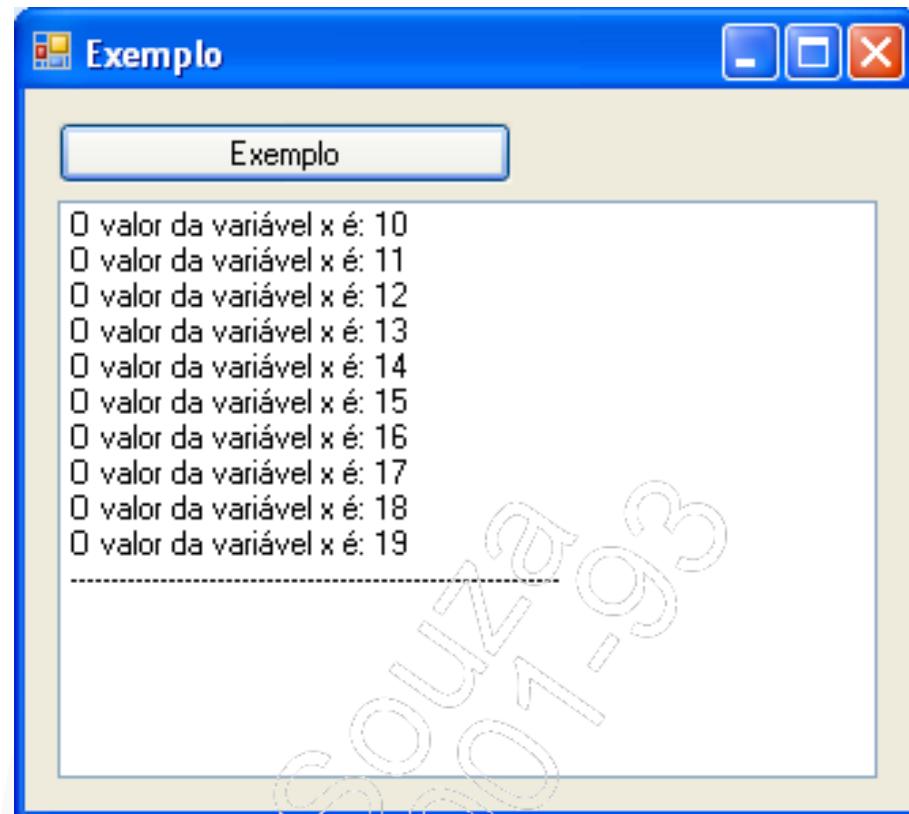
- A variável declarada no laço **for** pode ser utilizada apenas no próprio laço de repetição **for**;
 - A variável declarada fora do laço de repetição **for**, mas inicializada na instrução **for**, pode ser utilizada fora do laço de repetição.

No exemplo a seguir, mostramos a declaração de uma variável fora da instrução **for** cuja inicialização ocorre dentro da instrução **for**:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    int x;
    for (x = 10; x < 20; x++)
    {
        exemploListBox.Items.Add("O valor da variável x é: " + x);
    }
    exemploListBox.Items.Add(new string('-', 60));
}
```

C# - Módulo I

O resultado é o seguinte:



Isso nos mostra que as seções da instrução **for** não são dependentes e, por isso, não precisam operar sobre as mesmas variáveis. Quanto à expressão de iteração, ela não precisa, necessariamente, configurar ou incrementar algo, como mostramos no exemplo a seguir:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    int x = 6;
    for (int y = 1; x != 10; exemploListBox.Items.Add("O valor da variável x é: " + x))
    {
        x += y;
    }
    exemploListBox.Items.Add(new string('-', 60));
}
```

O resultado será o seguinte:



Ainda na instrução do laço de repetição **for**, podemos utilizar dois comandos, **break** e **continue**, os quais veremos a seguir. Geralmente, quando um desses comandos é utilizado, um teste **if** é executado dentro do laço de repetição **for**. Caso uma das condições verificadas seja verdadeira ou falsa, conforme o desejado na instrução, o laço de repetição ou a iteração serão finalizados e a próxima instrução do programa será executada.

5.2.4. Break

O comando **break** é utilizado para interromper um laço de repetição **while**, **do / while** ou **for** ou comando **switch / case** antes que ele seja completo. Feita a interrupção, a próxima instrução após o comando **break** é executada, caso haja alguma. Se existir mais de uma instrução **while**, **do / while**, **for** ou **switch / case** aninhadas, o comando **break** terá efeito sobre aquela que, entre elas, estiver no nível mais interno, ou seja, contida em todas as outras. Caso o comando **break** não esteja contido em nenhuma instrução ou comando a que ele se atribua, um erro ocorrerá no momento de compilação.



Também podemos utilizar **break** em instruções **foreach**, as quais, porém, não serão abordadas neste módulo do curso.

C# - Módulo I

O exemplo a seguir ilustra a utilização da instrução **break**:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    for (int x = 32; x < 2000; x *= 2)
    {
        if (x > 600)
        {
            //Interrompe a execução do for
            break;
        }
        exemploListBox.Items.Add("Valor de x é: " + x);
    }
    exemploListBox.Items.Add(new string(' ', 60));
}
```

O resultado será o seguinte:



5.2.5. Continue

O comando **continue** pode ser usado em laços de repetição **while**, **do / while** ou **for**, com a finalidade de voltar à execução da condição do laço de repetição ao qual ele se aplica. Assim como ocorre com o comando **break**, um erro durante a compilação é gerado caso o **continue** não esteja contido em um ambiente em que ele possa ser aplicado, ou seja, um laço de repetição. Outras características que os dois comandos compartilham são o fato de que, para instruções aninhadas, terão efeito sobre a que estiver no nível mais interno, e também a impossibilidade de definir o ponto final do comando em si, já que ele transfere o controle para outra parte do código.



Também podemos utilizar **continue** em instruções **foreach**, as quais, porém, não serão abordadas neste módulo do curso.

O exemplo a seguir ilustra a utilização da instrução **continue**:

```
while (!EOF)
{
    if (erro)
    {
        continue;
    }
}
```

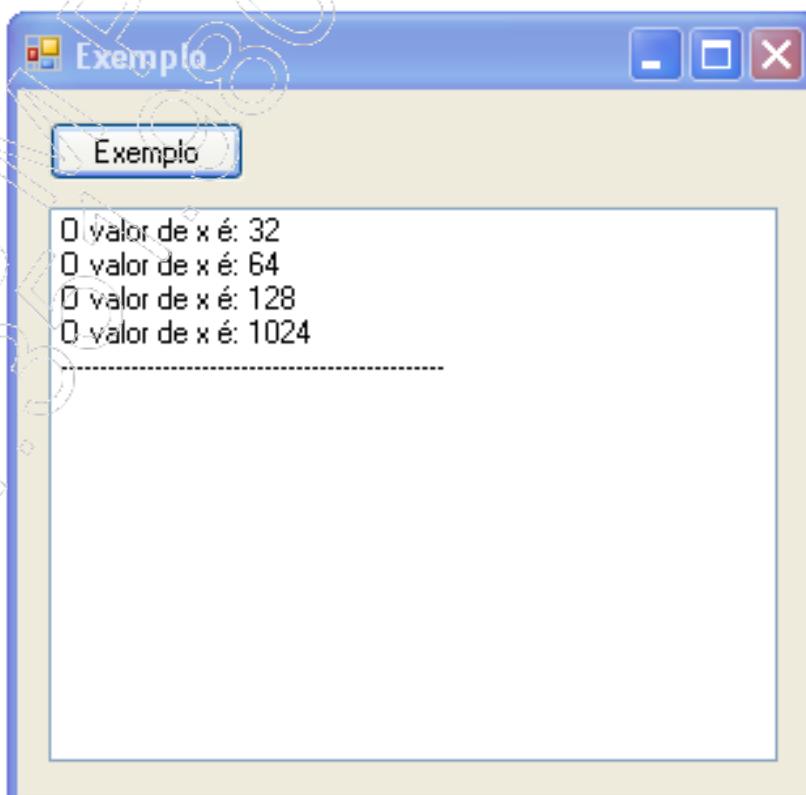
Quando esse código for executado, será verificado se há um erro. Se a condição for verdadeira, o próximo campo do arquivo será lido até que não seja o final de arquivo (**!EOF**). A instrução **continue** será utilizada para voltar ao laço de repetição e fazer com que os outros campos continuem a ser lidos até o final de arquivo. **EOF** simula uma variável de leitura de um recordset.

C# - Módulo I

O exemplo a seguir mostra outra utilização do comando **continue** com o laço de repetição **for**:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    for (int x = 32; x < 2000; x *= 2)
    {
        if (x == 256 || x == 512)
        {
            //Sempre que a condição if for atendida,
            //a execução volta para o if não executando
            //a linha de instrução
            continue;
        }
        exemploListBox.Items.Add("Valor de x é: " + x);
    }
    exemploListBox.Items.Add(new string('-', 60));
}
```

A compilação e a execução desse código resultarão no seguinte:



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Para decidirmos qual ação deverá ser tomada em determinada parte de um programa, solicitamos que este avalie uma expressão. Caso essa expressão seja avaliada como verdadeira, uma sequência de comandos será executada;
- Quando utilizamos uma instrução **if**, apenas expressões com resultados booleanos podem ser avaliadas;
- Quando trabalhamos com a instrução **if** e desejamos executar comandos nos casos em que a condição é avaliada como falsa, é necessário digitar a cláusula **else** após as instruções da condição verdadeira e, na linha abaixo, inserir o bloco de instruções a serem executadas;
- A instrução **switch** é um modo de simular a utilização de várias instruções **if** e pode somente verificar uma relação de igualdade. É indicada para avaliar uma única expressão para mais de um valor possível, deixando o programa mais legível e tornando sua execução mais eficiente;
- Quando não sabemos quantas vezes um determinado bloco de instruções precisa ser repetido, utilizamos o laço de repetição **while**. Com ele, a execução das instruções continuará enquanto uma determinada condição for verdadeira;
- A instrução **do / while** tem basicamente o mesmo funcionamento do **while**, porém, executa todos os comandos ao menos uma única vez, mesmo quando a condição não é verdadeira;
- Podemos utilizar o laço de repetição **for** quando sabemos exatamente quantas vezes um bloco de instruções deve ser repetido ou até que um limite preestabelecido seja alcançado.

5

Instruções de repetição

Teste seus conhecimentos

Carlos 77.351.9000-07-03



IMPACTA
EDITORA

1. Avalie o trecho de código a seguir e assinale a alternativa que contém o texto de label1 no final do processo:

```
int n = 0;  
label1.Text = "";  
while (n <= 10)  
{  
    label1.Text += n.ToString("00");  
    n += 2;  
}
```

- a) Ocorre um loop infinito.
- b) 00020406081012
- c) 020406081012
- d) 000204060810
- e) 0246810

2. Qual das instruções de repetição adiante será executada enquanto a condição for falsa?

- a) !while
- b) do / while
- c) for
- d) O C# não possui instrução de repetição executada enquanto a condição for falsa.
- e) Todas as instruções de repetição do C# serão executadas enquanto a condição for falsa.

3. Avalie o trecho de código a seguir e assinale a alternativa que contém o texto de label1 no final do processo:

```
int n = 12;
label1.Text = "";
while (n <= 10)
{
    n -= 2;
    label1.Text += n.ToString("00");
}
```

- a) Ocorre um loop infinito.
- b) 02
- c) 12100806040200
- d) vazio
- e) 100806040200

4. Avalie o trecho de código a seguir e assinale a alternativa que contém o texto de label1 no final do processo:

```
label1.Text = "";
for (int n = 0; n <= 10; n += 2)
{
    label1.Text += n.ToString("00");
}
```

- a) Ocorre um loop infinito.
- b) 00020406081012
- c) 020406081012
- d) 000204060810
- e) 0246810

5. Avalie o trecho de código a seguir e assinale a alternativa que contém o texto de label1 no final do processo:

```
int n = 0;  
label1.Text = "";  
do  
{  
    n += 2;  
    label1.Text += n.ToString("00");  
} while (n >= 10);
```

- a) Ocorre um loop infinito.
- b) 00020406081012
- c) 02
- d) 000204060810
- e) 0246810

5

Instruções de repetição Mãos à obra!

CarloS
77.357.9000-007-03



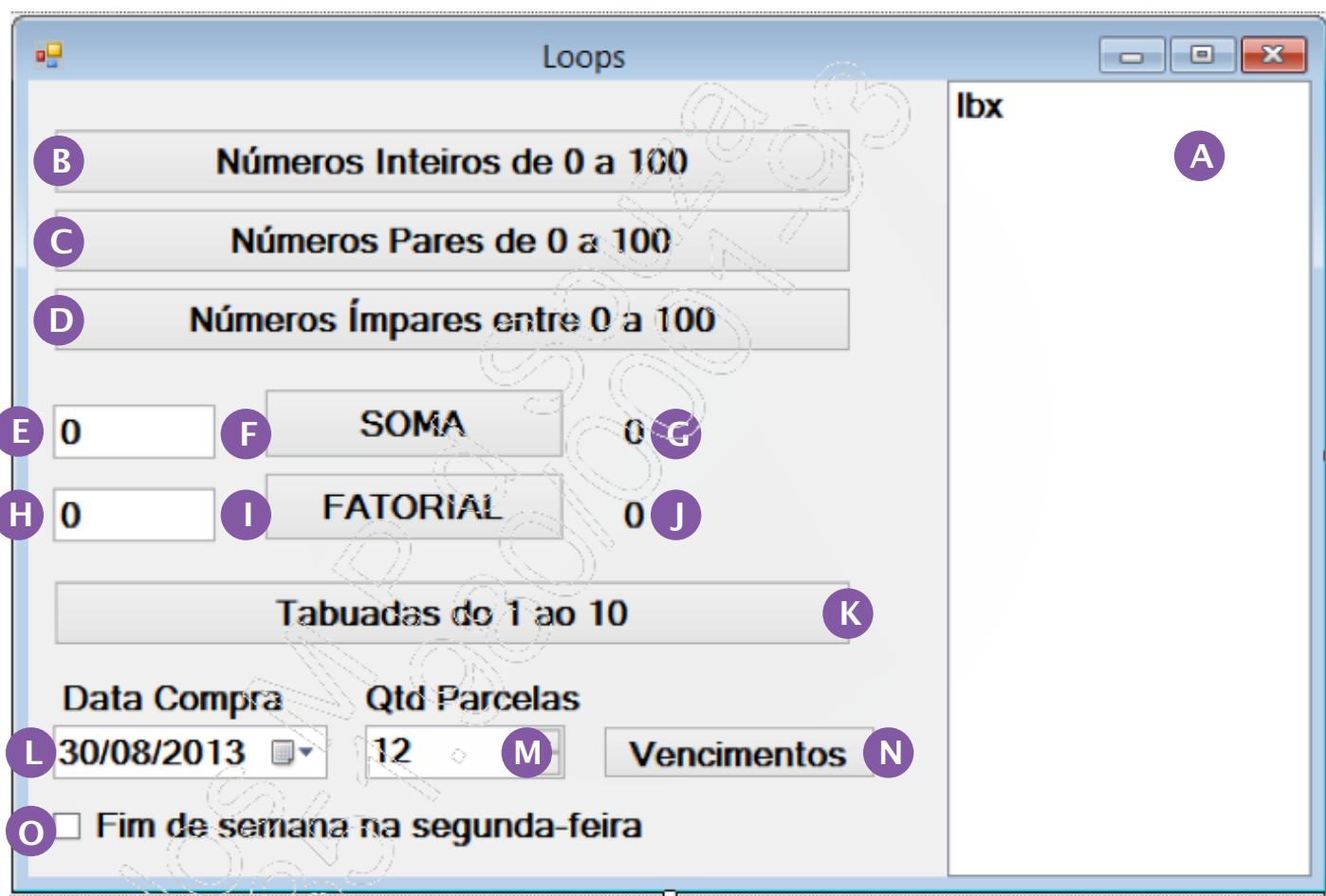
IMPACTA
EDITORA

Laboratório 1

A - Criando uma tela que realiza várias operações com loops

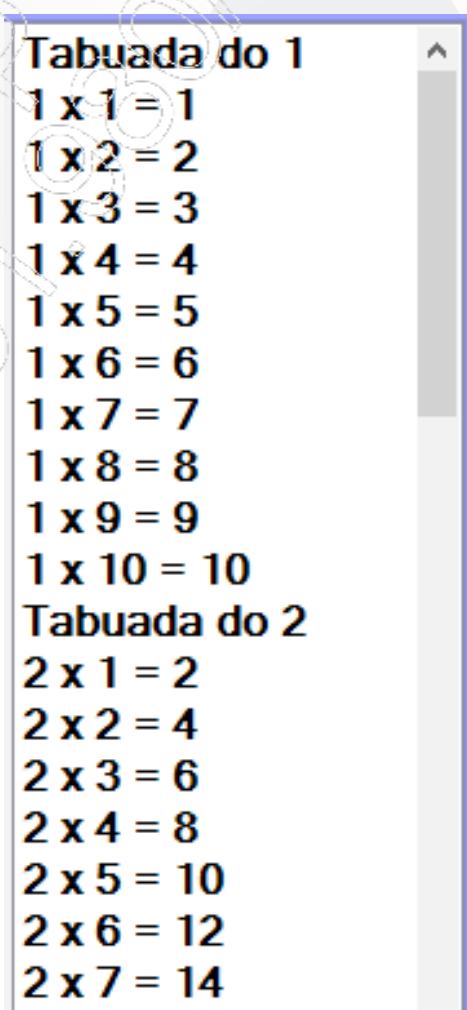
Neste laboratório, criaremos um projeto que executará diversas operações usando os loops **FOR** e **WHILE**. Cada botão será um exemplo diferente envolvendo os loops.

1. Crie o projeto **Loops** e monte a tela de acordo com as indicações a seguir:



- A - **ListBox** **Ibx**;
- B - **Button** **inteirosButton**: Deve inserir no **ListBox** (A) todos os números inteiros de 0 até 100;
- C - **Button** **paresButton**: Deve inserir no **ListBox** (A) todos os números pares de 0 até 100;
- D - **Button** **imparesButton**: Deve inserir no **ListBox** (A) todos os números ímpares entre 0 e 100;

- **E - TextBox somaTextBox;**
- **F - Button somaButton:** Deve mostrar no Label (G) a soma dos números inteiros entre 1 e o número digitado no TextBox (E). Por exemplo, se digitarmos o número 5 no campo E, o botão deve calcular $1 + 2 + 3 + 4 + 5 = 15$ e mostrar o resultado da soma no Label;
- **G - Label somaLabel;**
- **H - TextBox factorialTextBox;**
- **I - Button factorialButton:** Deve mostrar no Label (J) a multiplicação dos números inteiros entre 1 e o número digitado no TextBox (H). Por exemplo, se digitarmos o número 5 no campo E, o botão deve calcular $1 * 2 * 3 * 4 * 5 = 120$ e mostrar o resultado do cálculo no Label;
- **J - Label factorialLabel;**
- **K - Button tabuadasButton:** Deve mostrar no ListBox as tabuadas do 1 ao 10, como mostra a imagem a seguir:



- **L - DateTimePicker compraDtTime;**
- **M - NumericUpDown parcelasNumUpd;**
- **N - Button vencimentosButton:** Admitindo que a data informada em (L) é a data da compra, o botão deve mostrar no ListBox todas as datas de vencimento entendendo que os pagamentos serão mensais, ou seja, somar 1 mês na data da compra para cada vencimento (M);
- **O - CheckBox fimSemanaCheckBox:** Caso esta opção seja marcada, vencimentos ocorridos no sábado ou domingo deverão ser transferidos para segunda-feira.

Dica:

```
// armazena o conteúdo de um DateTimePicker em uma variável.  
DateTime dataCompra = compraDtTime.Value;  
// cria uma nova data somando 1 mês na data anterior  
DateTime dataVenc = dataCompra.AddMonths(1);  
// testa se o dia da semana da data é sábado.  
if (dataVenc.DayOfWeek == DayOfWeek.Saturday)  
    // Se sim, adiciona 2 dias nesta data  
    dataVenc = dataVenc.AddDays(2);
```

Arrays

6

- ✓ O que são arrays;
- ✓ Construção e instanciação de arrays;
- ✓ Conhecendo o tamanho de um array;
- ✓ Passando um array como parâmetro;
- ✓ Exemplos.



IMPACTA
EDITORA

6.1. O que são arrays?

Chamamos de array uma sequência não ordenada de elementos do mesmo tipo. Os arrays, que constituem um tipo especial de variável, contêm uma ou múltiplas dimensões e possuem valores que são identificados ou referenciados por um índice, o qual é um valor inteiro que determina a posição desses elementos dentro de um array. No .NET Framework, todos os índices começam pelo índice zero.

Um array pode ter elementos de qualquer tipo, inclusive outros arrays. Um array de arrays também é chamado de jagged array. Seus elementos são tipos-referência e seus valores iniciais são **null**.

6.2. Construção e instanciação de arrays

Para construir um array, é necessário especificar o nome do tipo de elemento, seguido por um par de colchetes e pelo nome da variável, conforme a seguinte sintaxe:

```
<tipo>[] <identificador>;
```

Podemos, também, criar uma instância de um array, utilizando, para isso, o operador **new**, seguido pelo nome do tipo de elemento e pelo tamanho do array entre colchetes, como vemos na sintaxe a seguir:

```
<tipo>[] <identificador> = new <tipo>[nº ocorrencias];
```

- **Exemplo 1: Array unidimensional**

```
int[] numeros = new int[10];  
// ou  
// int[] numeros = {0,0,0,0,0,0,0,0,0,0};
```

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

```
numeros[0] = 18;  
numeros[1] = 27;  
numeros[2] = 10;  
numeros[3] = 0;  
numeros[4] = 45;  
numeros[5] = 22;  
numeros[6] = 34;  
numeros[7] = 56;  
numeros[8] = 76;  
numeros[9] = 5;
```

0	1	2	3	4	5	6	7	8	9
18	27	10	0	45	22	34	56	76	5

Observação: O primeiro elemento de um array está sempre na posição ZERO.

- **Exemplo 2: Array bidimensional**

```
// int[,] numeros = new int[3, 5];  
// ou  
int[,] numeros = { { 17, 22, 12, 44, 34 },  
                   { 23, 33, 12, 31, 45},  
                   { 2, 45, 35, 1, 22} };
```

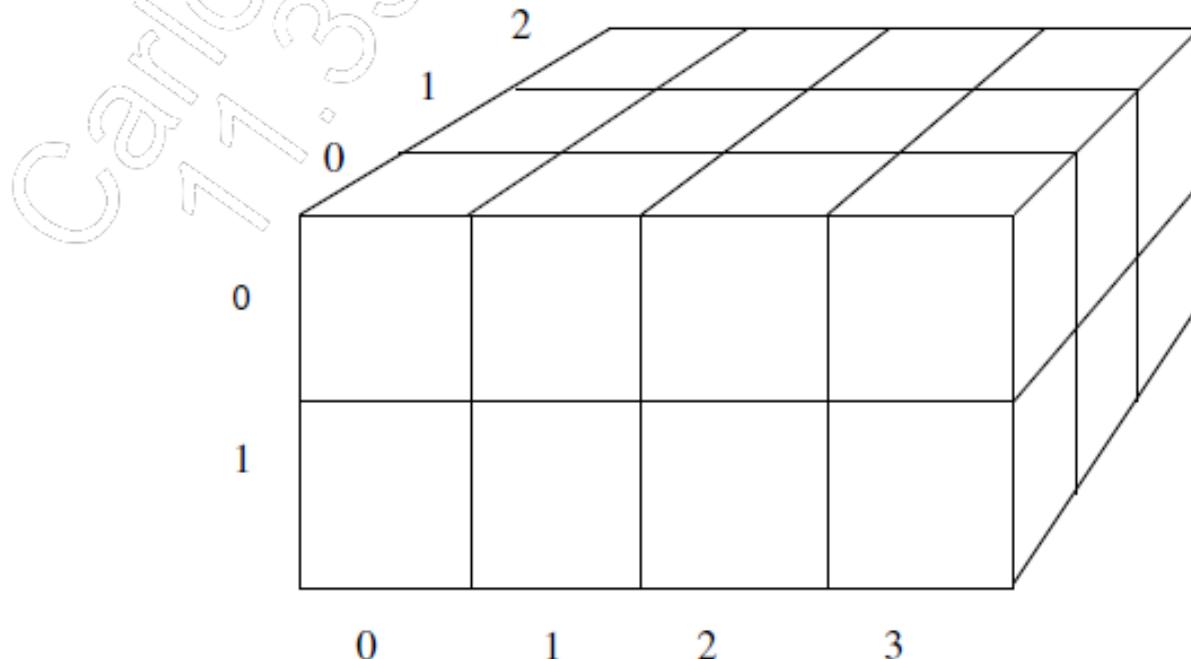
	0	1	2	3	4
0	17	22	12	44	34
1	23	33	12	31	45
2	2	45	35	1	22

Responda qual o conteúdo dos elementos:

- a) numeros[1,3] _____
- b) numeros[3,1] _____
- c) numeros[2,4] _____
- d) numeros[0,2] _____
- e) numeros[2,0] _____

- **Exemplo 3: Array tridimensional**

```
int[, ,] numeros = new int[2, 4, 3];
```



6.3. Conhecendo o tamanho de um array

Para saber o tamanho de um array, utilizamos sua propriedade **Length**. Por meio de tal propriedade, é possível iterar por todos os elementos de um array utilizando uma instrução **for**. Na instrução **for**, a iteração é sempre feita do índice **0** para o índice **Length - 1**, nos arrays de uma única dimensão.

Há, ainda, uma instrução chamada **foreach**, que permite iterar por todos os elementos de um array diretamente. Ela declara uma variável que recebe automaticamente o valor de cada elemento do array. A instrução **foreach** indica a intenção do código de maneira direta e torna desnecessário utilizar toda a estrutura **for**.

Por essas características, a instrução **foreach** é a maneira preferida para iterar por um array. Ainda assim, há algumas situações em que a instrução **for** é mais adequada:

- Para iterar apenas por parte de um array;
- Para iterar de trás para frente;
- Quando é necessário que o corpo do laço saiba não apenas o valor, mas o índice do elemento;
- Quando houver intenção de alterar os elementos do array, já que a variável de iteração de **foreach** é uma cópia somente para leitura de cada um dos elementos que compõem o array.

Vejamos um exemplo utilizando **for**:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    int[] numeros = { 0, 50, 100, 150, 200, 250, 300, 350, 400 };

    for (int i = 0; i <= numeros.Length - 1; i++)
    {
        exemploListBox.Items.Add(
            "numeros[" + i + "] = " + numeros[i]);
    }
    exemploListBox.Items.Add(new string('-', 40));
}
```

É possível, também, percorrer os elementos do array através de um laço **foreach**:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    int[] numeros = { 0, 50, 100, 150, 200, 250, 300, 350, 400 };

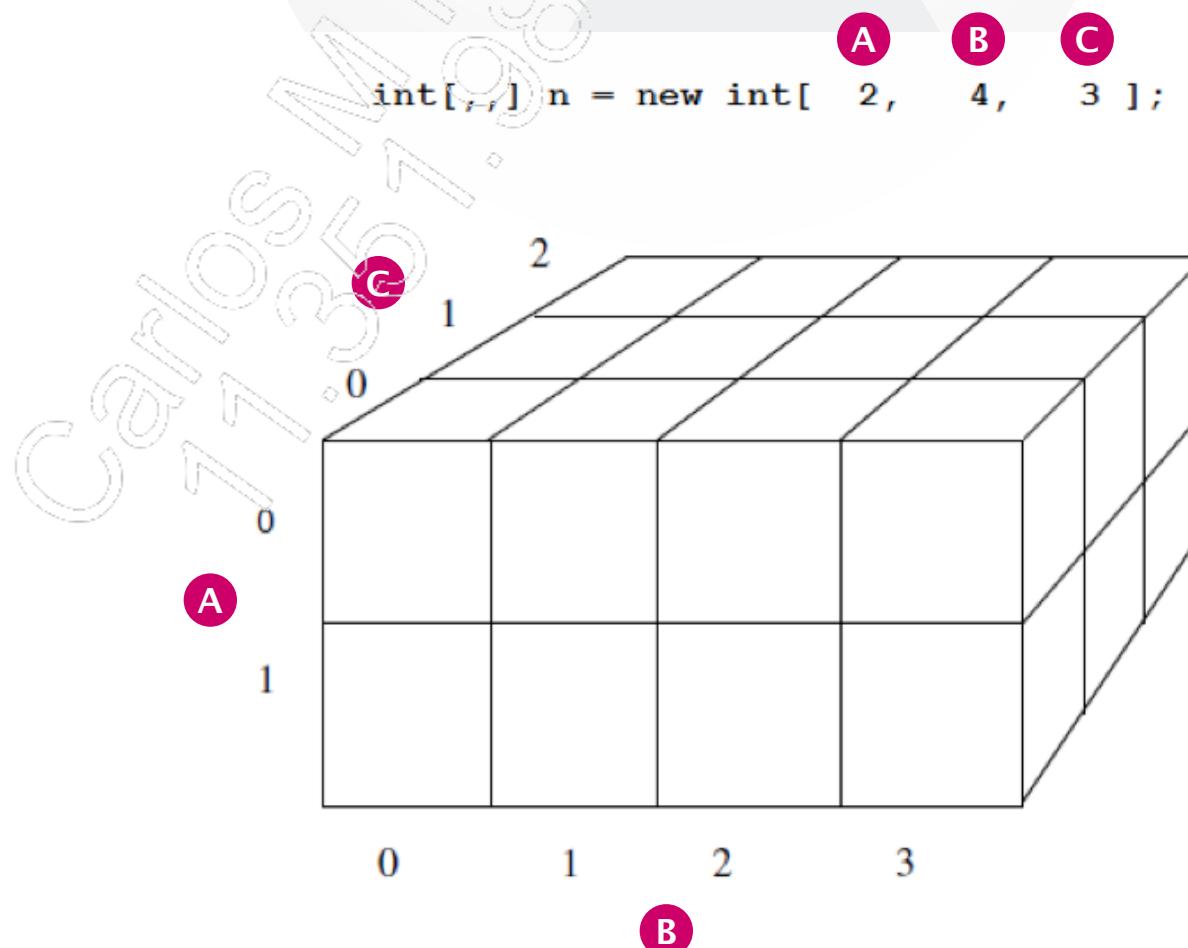
    foreach (int i in numeros)
    {
        exemploListBox.Items.Add("numeros: " + i);
    }
    exemploListBox.Items.Add(new string('-', 40));
}
```

6.3.1. Arrays com várias dimensões

Em um array com várias dimensões, a propriedade **Length** retorna com a quantidade total de elementos do array, o que não tem muita utilidade:

```
int[,] numeros = new int[3, 3];
// vai retornar 9 que é a quantidade total de elementos
label1.Text = numeros.Length.ToString();
```

Neste caso, para sabermos o tamanho de cada dimensão, vamos usar o método **GetLength(dimensão)**.



- A - Primeira dimensão ou dimensão zero: **numeros.GetLength(0)** retorna 2;
- B - Segunda dimensão ou dimensão 1: **numeros.GetLength(1)** retorna 4;
- C - Terceira dimensão ou dimensão 2: **numeros.GetLength(2)** retorna 3.

Há alguns pontos que merecem ser destacados com relação ao uso de arrays:

- Em qualquer dimensão de um array, o primeiro elemento tem índice 0;
- Todos os elementos de um array serão do mesmo tipo;
- As dimensões de um array são determinadas em tempo de compilação, não é possível aumentar ou diminuir o tamanho de um array durante a execução da aplicação;
- Para saber o total de dimensões de um array, deve-se utilizar a propriedade **Rank**:

```
int[, ,] numeros = new int[2, 3, 4];
// vai retornar 3
label1.Text = numeros.Rank.ToString();
```

6.4. Passando um array como parâmetro

Um array pode ser passado e recebido como parâmetro. Tal característica nos permite escrever métodos que podem receber como parâmetros um número qualquer de argumentos, de qualquer tipo. Isso pode ser útil em algumas situações, como aquelas em que houver a necessidade de um método que determine o valor mínimo em um conjunto de valores passados como parâmetros.

Vejamos um exemplo:

```
private decimal CalcularMedia(decimal[] valores)
{
    decimal soma = 0;

    for (int i = 0; i <= valores.GetUpperBound(0); i++)
    {
        soma += valores[i];
    }
    return soma / valores.Length;
}
```

O código da utilização desse método é o seguinte:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    decimal[] numeros = { 15, 20, 25, 30, 150 };

    for (int i = 0; i <= numeros.GetUpperBound(0); i++)
    {
        exemploListBox.Items.Add(numeros[i]);
    }
    exemploListBox.Items.Add(new string('-', 20));

    exemploListBox.Items.Add(CalcularMedia(numeros));
}
```

Desse modo, evitamos o uso de sobrecargas (utilizadas para declarar dois ou mais métodos no mesmo escopo com nomes iguais). Isso torna necessário, porém, escrever um código a mais para preencher o array passado como parâmetro.

É importante destacar, ainda, que quaisquer alterações realizadas nos valores do método que receber um array como parâmetro serão refletidas no array original, já que os arrays são passados como referência.

6.4.1. Palavra-chave params

A palavra-chave **params** funciona como um modificador dos parâmetros de array. Por meio dela, é possível definir um parâmetro de método que utiliza qualquer número de argumentos.

Vejamos um exemplo:

```
private decimal CalcularMedia(params decimal[] valores)
{
    decimal soma = 0;

    for (int i = 0; i <= valores.GetUpperBound(0); i++)
    {
        soma += valores[i];
    }
    return soma / valores.Length;
}
```

O código da utilização desse método é o seguinte:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    exemploListBox.Items.Add("Média: " +
        CalcularMedia(15, 20, 25, 30, 150));
}
```

É necessário ressaltar alguns aspectos sobre a utilização da palavra-chave **params** na declaração de um método:

- Não é possível utilizar **params** em arrays multidimensionais;
- Só é possível utilizar uma palavra **params** por declaração;
- Após a palavra **params**, não podemos acrescentar outros parâmetros;
- Um método não **params** sempre terá prioridade sobre um método **params**.

6.4.1.1. Params object[]

Além da possibilidade de passar uma quantidade diversa de argumentos a um método, o C# permite também a diversidade de tipos de argumentos. Para isso, é utilizada a classe base **object**. Baseando-se no fato de que o compilador pode gerar um código que transforma tipos-valor em objetos, torna-se possível utilizar um array de parâmetros do tipo **object** para declarar um método que aceite argumentos de qualquer tipo.

A seguir, podemos visualizar um exemplo:

```
private void ExibirDados(params object[] valores)
{
    foreach (object i in valores)
    {
        exemploListBox.Items.Add(i);
    }
}
```

O código da utilização desse método é o seguinte:

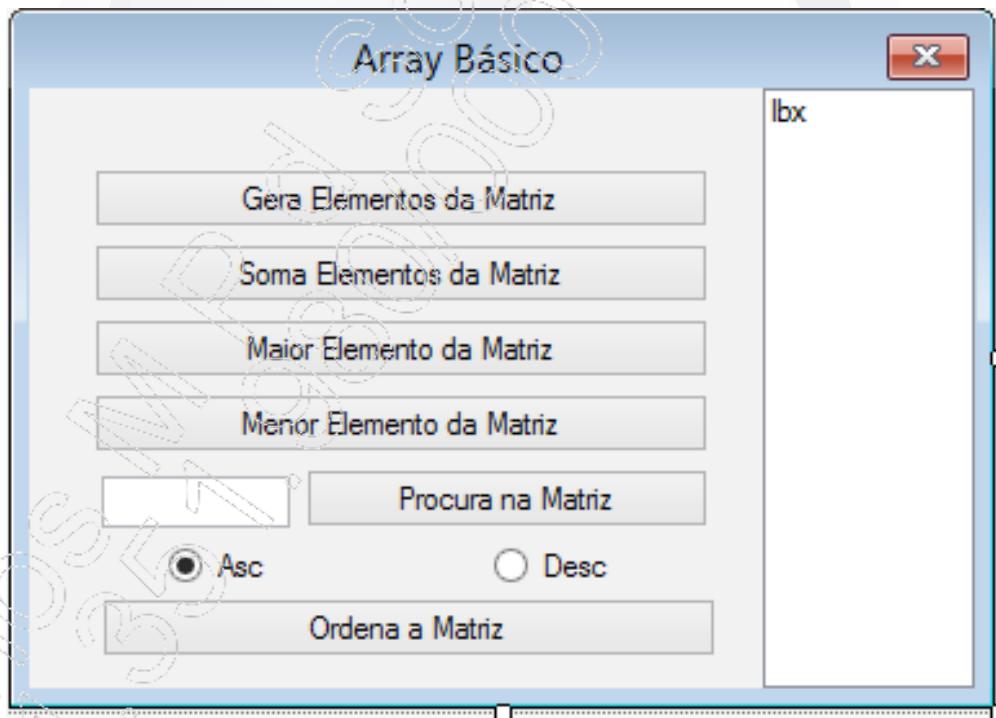
```
private void exemploButton_Click(object sender, EventArgs e)
{
    {
        ExibirDados("Jatobá", 3.45m, "Cajamanga", 2.47f, 200);
    }
}
```

6.5. Exemplos

A seguir, trabalharemos com alguns projetos que exemplificam o uso de arrays.

6.5.1. Exemplo 1

Abra o projeto que está na pasta **Cap_06\1_ArrayBasico_Ex1**:



1. Declare a variável **array** visível em todo o formulário:

```
public partial class Form1 : Form
{
    // variáveis para uso de todos os métodos da classe
    // devem ser declaradas aqui
    // declaração do array: tipo[] nome = new tipo[ tamanho ]
    int[] numeros = new int[10];
    // ou
    // int[] numeros = { 0,0,0,0,0,0,0,0,0,0 };
```

2. Declare a variável para gerar números aleatórios que irão preencher os elementos do array;

```
public partial class Form1 : Form
{
    // variáveis para uso de todos os métodos da classe
    // devem ser declaradas aqui
    // declaração do array: tipo[] nome = new tipo[ tamanho ]
    int[] numeros = new int[10];
    // ou
    // int[] numeros = { 0,0,0,0,0,0,0,0,0,0 };

    // gerador de números aleatórios
    Random rn = new Random();
```

3. Utilize o evento **Click** do botão **Gera Elementos da Matriz**:

```
private void geraButton_Click(object sender, EventArgs e)
{
    // apaga todos os itens do ListBox
    lbx.Items.Clear();
    // loop para percorrer todos os elementos do ListBox
    for (int i = 0; i < numeros.Length; i++)
    {
        // gera número aleatório no intervalo de 1 a 60
        // e armazena no array
        numeros[i] = rn.Next(1, 61);
        // mostra o elemento do array no ListBox
        lbx.Items.Add(numeros[i]);
    }
}
```

4. Utilize o evento Click do botão Soma Elementos da Matriz:

```
private void somaButton_Click(object sender, EventArgs e)
{
    // variável para armazenar a soma dos elementos
    int soma = 0;
    // loop para percorrer os elementos do array
    for (int i = 0; i < numeros.Length; i++)
    {
        // soma na variável o elemento atual do array
        soma += numeros[i];
    }

    // mostra no ListBox o valor da soma
    lbx.Items.Add("-----");
    lbx.Items.Add("Soma = " + soma);
    lbx.Items.Add("-----");
}
```

5. Utilize o evento Click do botão Maior Elemento da Matriz:

```
private void maiorButton_Click(object sender, EventArgs e)
{
    // define o primeiro elemento como sendo o maior
    int maior = numeros[0];
    // loop para percorrer o array a partir do
    // segundo elemento
    for (int i = 1; i < numeros.Length; i++)
    {
        // se o elemento atual for maior que a variável maior
        if (numeros[i] > maior)
        {
            // trocar o conteúdo da variável
            maior = numeros[i];
        }
    }
    // mostrar o maior elemento
    lbx.Items.Add("-----");
    lbx.Items.Add("Maior = " + maior);
    lbx.Items.Add("-----");
}
```

6. Crie o evento **Click** do botão **Menor Elemento da Matriz**, tomando como base o passo anterior;

7. Utilize o evento **Click** do botão **Procura na Matriz**. Ele deve procurar no array o número digitado no **TextBox**.

```
private void procuraButton_Click(object sender, EventArgs e)
{
    // número que queremos procurar na matriz
    int n = Convert.ToInt32(tbxNum.Text);
    // posição do número dentro da matriz
    int pos = -1;

    for (int i = 0; i < numeros.Length; i++)
    {
        // se o número que está na posição i da matriz for igual
        // ao número que estamos procurando
        if (numeros[i] == n)
        {
            // colocar na variável pos a posição atual
            pos = i;
            // sair do loop - só pode ser usado dentro de for ou while
            break;
        }
    }
    if (pos < 0) // não encontrou
        MessageBox.Show("Não encontrado");
    else
        // seleciona no listBox o item que estiver na posição pos
        lbx.SelectedIndex = pos;
}
```

C# - Módulo I

8. Teste o exemplo até este ponto. Então, altere o código como mostra o trecho a seguir:

```
private void procuraButton_Click(object sender, EventArgs e)
{
    // número que queremos procurar na matriz
    int n = Convert.ToInt32(tbxNum.Text);
    // posição do número dentro da matriz
    int pos = Array.IndexOf(numero, n);
    //int pos = -1;

    //for (int i = 0; i < numero.Length; i++)
    //{
    //    // se o número que está na posição i da matriz for igual
    //    // ao número que estamos procurando
    //    if (numero[i] == n)
    //    {
    //        // colocar na variável pos a posição atual
    //        pos = i;
    //        // sair do loop - só pode ser usado dentro de for ou while
    //        break;
    //    }
    //}
    if (pos < 0) // não encontrou
        MessageBox.Show("Não encontrado");
    else
        // seleciona no listBox o item que estiver na posição pos
        lbox.SelectedIndex = pos;
}
```

A classe Array possui várias rotinas úteis para a manipulação de arrays.

9. Crie o evento **Click** do botão **Ordena a Matriz**. Ele deve ordenar os elementos e exibi-los novamente no ListBox, já os RadioButtons definem como será feita a ordenação.

Para ordenar na descendente são utilizados recursos que ainda não vimos, como expressões lambda, LINQ, interfaces e classes. Portanto, para concluir este exemplo, usaremos o seguinte artifício:

A classe Array possui o método **Sort()** que ordena o array na ascendente se for usado na sua forma mais simples:

```
// ordena o array na ascendente  
Array.Sort(numeros);
```

Essa classe também possui um método chamado **Reverse()**, que inverte as posições dos elementos. Ele não os ordena, apenas os inverte, ou seja, o último passa a ser o primeiro, o penúltimo passa a ser o segundo e assim por diante. Ora, se ordenarmos na ascendente e depois invertermos, teremos ordenado os elementos na descendente.

```
// ordena o array na ascendente  
Array.Sort(numeros);  
// se for para ordenar na descendente, inverte  
if (descRadio.Checked) Array.Reverse(numeros);
```

Agora conclua o evento **Click** deste botão.

6.5.2. Exemplo 2

Abra o projeto da pasta **Cap_06\2_Anima_Ex2** fornecida pelo instrutor.



C# - Módulo I

Na pasta **bin\Debug**, onde está o executável da aplicação, foram copiadas 11 imagens de olhos que, se exibidas em intervalos de tempo diferentes, criarião um efeito de olhos girando. Para isso, devemos ter os seguintes elementos:

- Um array contendo os nomes dos arquivos das imagens, na ordem na qual deverão ser exibidas;

```
public partial class Form1 : Form
{
    string[] anima = {"ANIMATAC 001 ICO", "ANIMATAC 002 ICO",
                      "ANIMATAC 003 ICO", "ANIMATAC 004 ICO",
                      "ANIMATAC 005 ICO", "ANIMATAC 006 ICO",
                      "ANIMATAC 007 ICO", "ANIMATAC 008 ICO",
                      "ANIMATAC 009 ICO", "ANIMATAC 010 ICO",
                      "ANIMATAC 011 ICO"};
```

- Uma variável para apontar qual elemento do array deverá ser mostrado na tela;

```
public partial class Form1 : Form
{
    string[] anima = {"ANIMATAC 001 ICO", "ANIMATAC 002 ICO",
                      "ANIMATAC 003 ICO", "ANIMATAC 004 ICO",
                      "ANIMATAC 005 ICO", "ANIMATAC 006 ICO",
                      "ANIMATAC 007 ICO", "ANIMATAC 008 ICO",
                      "ANIMATAC 009 ICO", "ANIMATAC 010 ICO",
                      "ANIMATAC 011 ICO"};
    // começa no zero porque já é a figura mostrada no momento
    int i = 0;
```

- Um timer para incrementar a variável **i** e mostrar a figura correspondente. As propriedades são:

Enabled: true;

Interval: 200.

Crie o evento Tick:

```
private void timer1_Tick(object sender, EventArgs e)
{
    // incrementa o índice do array
    // i++;
    // verifica se passou do último elemento
    // if (i >= anima.Length) i = 0;
    // idem anterior
    if (++i >= anima.Length) i = 0;
    // lê a figura do arquivo e mostra no PictureBox
    pictureBox1.Load(anima[i]);
}
```

! Como as figuras estão no mesmo diretório do EXE, não precisa indicar o caminho.

- O botão Liga/Desliga deve parar ou reiniciar a animação. Para isso, é necessário fazer o seguinte: se a propriedade **Enabled** do timer estiver **true**, mude para **false**. Caso contrário, mude para **true**;
- Crie um método para o evento **ValueChanged** da **TrackBar**. Para alterar a velocidade da animação, devemos alterar a propriedade **Interval** do timer. Quanto menor o **Interval**, mais rápido; quanto maior, mais lento. Conclua este método colocando na propriedade **Interval** do timer o conteúdo da propriedade **Value** da **TrackBar**.

6.5.3. Exemplo 3

Abra o projeto da pasta **Cap_06\3_Signos_Ex3** fornecida pelo instrutor.



Na pasta **bin\Debug**, onde está o executável da aplicação, foram copiados 12 ícones dos signos. Quando clicarmos no botão **Troca**, deverá ser exibido o ícone e o nome do signo seguinte. Para isso, precisaremos dos seguintes elementos:

- Um array contendo os nomes dos arquivos das figuras, na ordem na qual deverão ser exibidas, e também os nomes dos signos, na mesma ordem;

```
public partial class Form1 : Form
{
    /*
     * signos =
     * {
     *     {"Aries", "Aries.Bmp"}, {"Touro", "Touro.Bmp"}, {"Gêmeos", "Gemeos.Bmp"}, {"Câncer", "Cancer.Bmp"}, {"Leão", "Leao.Bmp"}, {"Virgem", "Virgem.Bmp"}, {"Libra", "Libra.Bmp"}, {"Escorpião", "Escorpiao.Bmp"}, {"Sagitário", "Sagitaro.Bmp"}, {"Peixes", "Peixes.Bmp"}, {"Capricórnio", "Capricornio.Bmp"} };
    */
    string[,] signos = {{"Aquário 21/01 a 19/02", "Peixes 20/02 a 20/03", "Áries 21/03 a 20/04", "Touro 21/04 a 20/05", "Gêmeos 21/05 a 20/06", "Câncer 21/06 a 21/07", "Leão 22/07 a 22/08", "Virgem 23/08 a 22/09", "Libra 23/09 a 22/10", "Escorpião 23/10 a 21/11", "Sagitário 22/11 a 21/12", "Capricórnio 22/12 a 20/01"}, {"Aquario.Bmp", "Peixes.Bmp", "Aries.Bmp", "Touro.Bmp", "Gemeos.Bmp", "Cancer.Bmp", "Leao.Bmp", "Virgem.Bmp", "Libra.Bmp", "Escorpiao.Bmp", "Sagitaro.Bmp", "Capricornio.Bmp"}};
}
```

- Uma variável para apontar qual elemento do array deverá ser mostrado na tela. Ela deve ser declarada para todo o Form (fora de método) para que mantenha o seu valor;

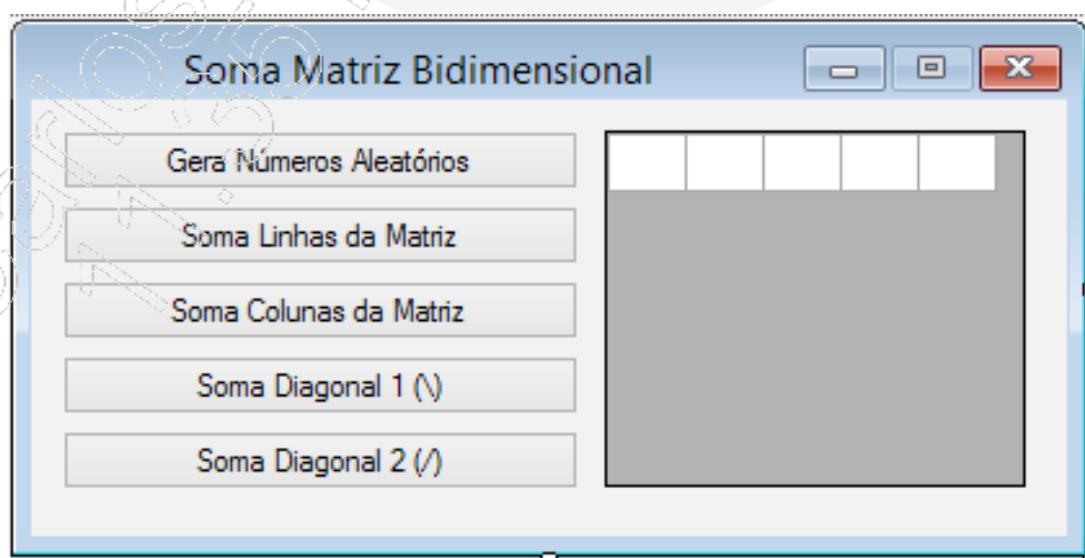
```
int i = 0;
```

- Complete as lacunas do evento Click do botão Troca:

```
private void btnTroca_Click(object sender, EventArgs e)
{
    // testar se ultrapassou o último signo
    // if (++i >= 12) i = 0;
    if (++i >= signos.GetLength(1)) i = 0;
    // os nomes dos signos estão na linha zero da matriz
    lblSigno.Text = signos[____, ____];
    // os nomes dos arquivos estão na linha 1 da matriz
    pbxSigno.Load(signos[____, ____]);
}
```

6.5.4. Exemplo 4

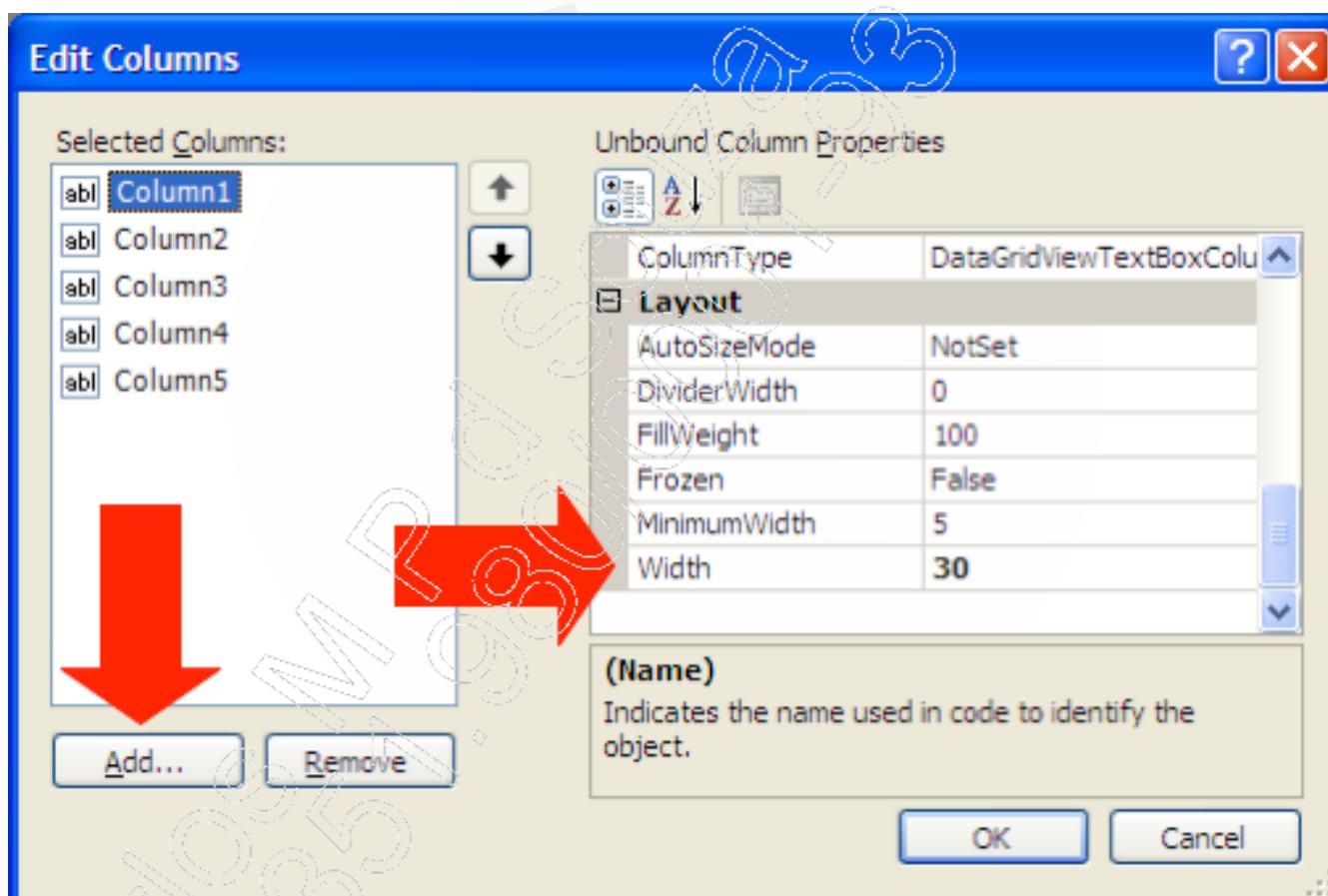
Abra o projeto que está na pasta **Cap_06\4_SomaMatriz_Ex4** fornecida pelo instrutor.



1. Configure o **DataGridView** de acordo com as propriedades indicadas e considerando as observações adiante:

- **Propriedades**

- **ColumnHeadersVisible = False;**
- **RowHeadersVisible = False**
- **RowHeadersWidthSizeMode = DisableResizing;**
- **RowDefaultCellStyle -> Alignment = MiddleRight;**
- **RowTemplate -> ReadOnly = True; Resizable = false;**
- **Columns (...);**



- Adicione cinco colunas e altere a propriedade **Width** de todas elas para 30;
- **Resizable = false** para todas as colunas.

- **Observações**

- As linhas e as colunas do **DataGridView** são numeradas a partir de zero;
- Para adicionar uma linha no **DataGridView**, utilize este código:

```
// dataGridView1.Rows.Add();
// ou
dataGridView1.RowCount = qtdLinhas
```

- Para colocar um dado em uma célula do grid, utilize o código a seguir:

```
dgvNumeros.Rows[linha].Cells[coluna] = vlrQqrTipo;
// OU
dgvNumeros[linha, coluna] = vlrQqrTipo;
```

2. Declare um array 3 por 3 de números inteiros com visibilidade para todos os métodos da classe. No mesmo local, declare também uma variável para manipular números randômicos;

```
public partial class Form1 : Form
{
    int[,] numeros = { { 0,0,0 },
                      { 0,0,0 },
                      { 0,0,0 } };

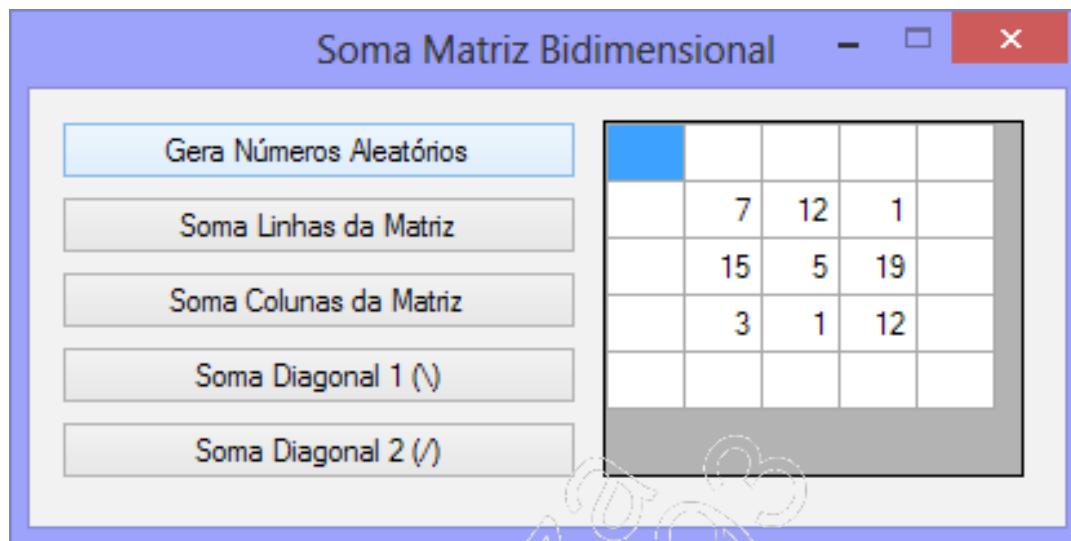
    Random rn = new Random();
```

3. Crie um método para o evento **Load** do formulário. Ele será executado toda vez que o formulário for carregado na memória;

```
private void Form1_Load(object sender, EventArgs e)
{
    // Adiciona 5 linhas no DataGridView
    dgvNumeros.RowCount = 5
}
```

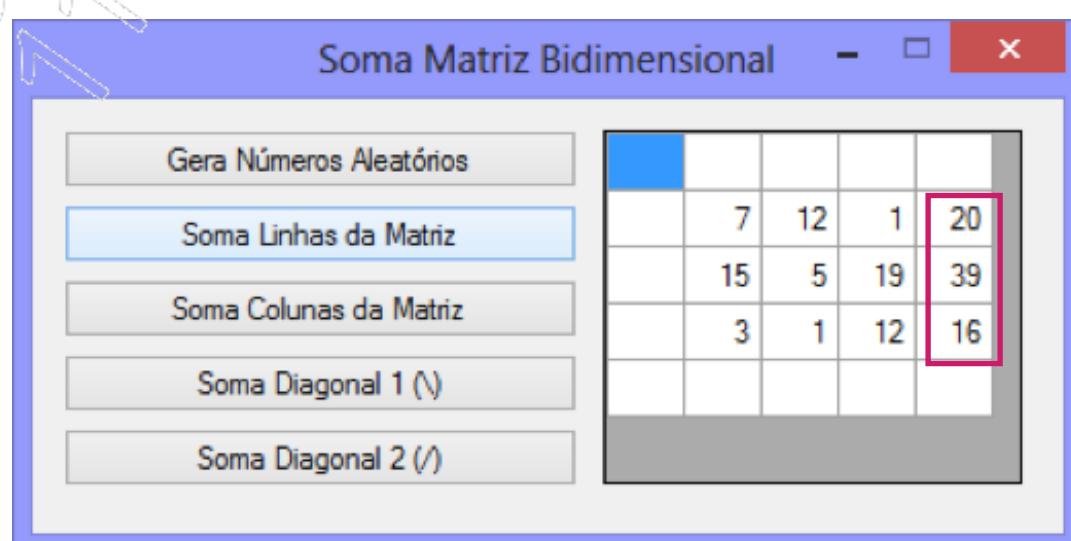
C# - Módulo I

4. Crie um método para o evento **Click** do botão **btnGeraNum**. Ele deve preencher o array com números aleatórios e exibi-los no grid, como mostra a imagem a seguir:

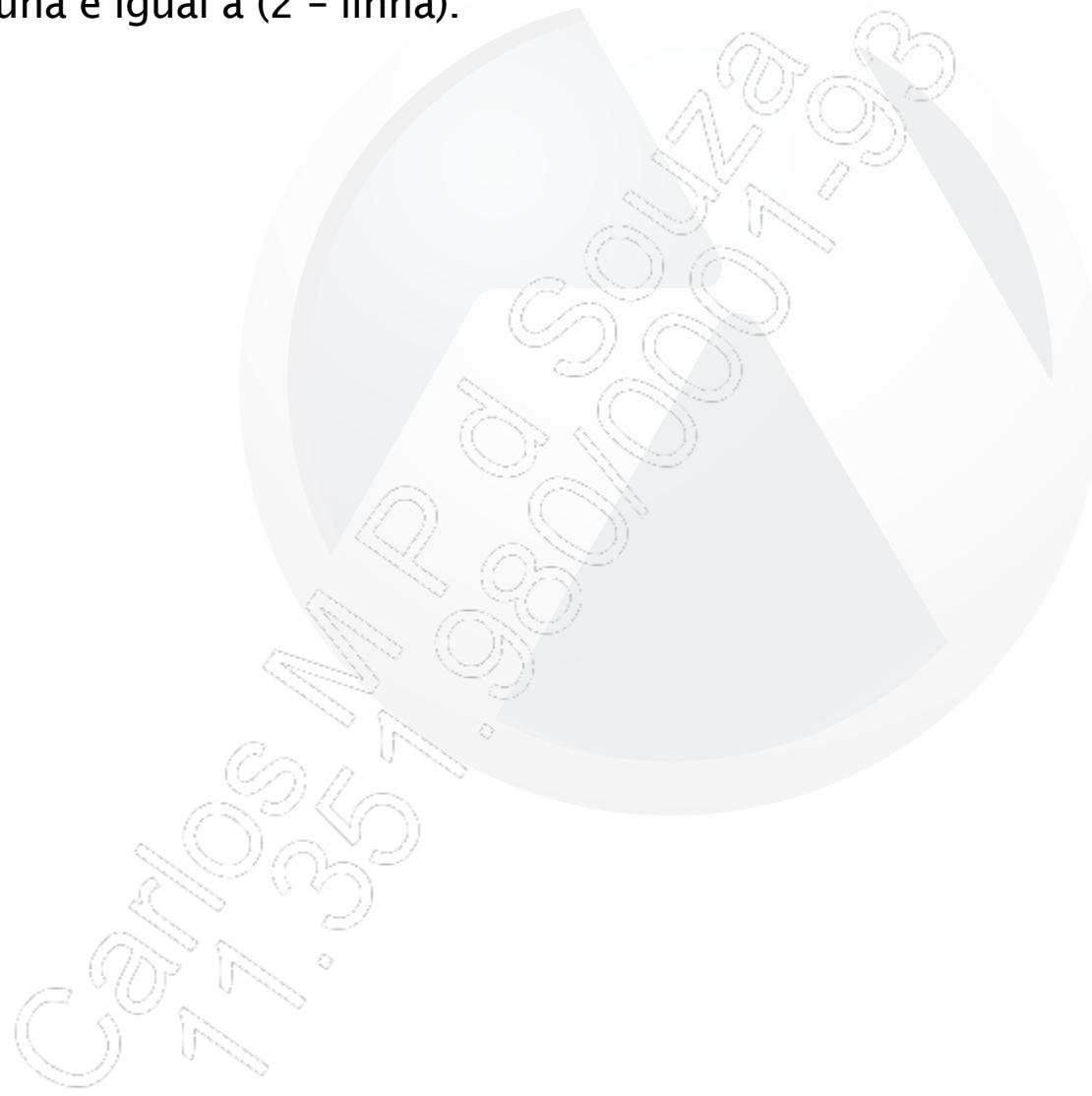


```
private void btnGeraNum_Click(object sender, EventArgs e)
{
    for (int lin = 0; lin < 3; lin++)
    {
        for (int col = 0; col < 3; col++)
        {
            numeros[l, c] = rn.Next(50);
            dgvNumeros[col + 1, lin + 1].Value = numeros[lin, col];
        }
    }
}
```

5. Crie um método para o evento **Click** do botão **btnSomaLinhas**. Ele deve somar cada uma das linhas do array e apresentar o resultado na última coluna do grid;



6. Crie um método para o evento **Click** do botão **btnSomaColunas**. A soma de cada coluna deve sair na linha número 4 do **DataGridView**;
7. Crie um método para o evento **Click** do botão **btnSomaD1**. A soma desta diagonal deve ser exibida na linha 4 e coluna 4 do grid. Lembre-se que os elementos desta diagonal estão nas posições **0,0**, **1,1** e **2,2**, logo, o número da linha é igual ao número da coluna;
8. Crie um método para o evento **Click** do botão **btnSomaD2**. A soma desta diagonal deve ser exibida na linha 4 e coluna 0 do grid. Lembre-se de que os elementos desta diagonal estão nas posições **0,2**, **1,1** e **2,0**, ou seja, o número da coluna é igual a $(2 - \text{linha})$.



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Um array é uma sequência não ordenada de elementos do mesmo tipo. Ele, que é um tipo especial de variável, contém múltiplas dimensões e seus valores são identificados por um índice;
- Os arrays podem ter várias dimensões de dados, o que os torna perfeitamente adequados para armazenar conjuntos de informações relacionadas;
- A maneira preferida para iterar por um array é utilizando a instrução **foreach**;
- Um array pode ser passado e recebido como parâmetro, o que é útil quando for necessário um método que determine o valor mínimo em um conjunto de valores passados como parâmetros.

6

Arrays

Teste seus conhecimentos

Carlos M. Gómez Souza
77.357.900/0007-93



IMPACTA
EDITORA

1. Quais valores serão mostrados em label1, label2 e label3 respectivamente?

```
int[] numeros = { 17, 25, 8, 21, 13, 19, 32, 14 };
label1.Text = (numeros[6] / numeros[2]).ToString();
label2.Text = (numeros[2] / 2).ToString();
label3.Text = numeros.Length.ToString();
```

- a) 0.76, 12.5, 8
- b) 0.76, 12.5, 7
- c) 4, 4, 7
- d) 4, 4, 8
- e) 4, 5, 8

2. Quais valores serão mostrados em label1, label2 e label3 respectivamente?

```
int[] numeros = new int[8];

for (int i = 0; i < numeros.Length; i++)
{
    numeros[i] = 3 * i + 1;
}
label1.Text = numeros[6].ToString();
label2.Text = numeros[3].ToString();
label3.Text = numeros[2].ToString();
```

- a) 22, 13, 10
- b) 12, 32, 34
- c) 20, 11, 8
- d) 19, 10, 7
- e) 0,0,0

3. Quais valores serão mostrados em label1, label2 e label3 respectivamente?

```
int[] numeros = new int[8];
    for (int i = 0; i < numeros.Length; i++)
    {
        numeros[i] = numeros.Length - i;
    }
    label1.Text = numeros[6].ToString(); // (1)
    label2.Text = numeros[3].ToString(); // (2)
    label3.Text = numeros[2].ToString(); // (3)
```

- a) 2, 5, 6
- b) 6, 5, 2
- c) 1, 2, 3
- d) 1, 4, 5
- e) 3, 2, 1

4. Em quais posições da matriz estão os elementos F, H e J?

```
string[,] letras = new string[3,4];
```

A	B	C	D
E	F	G	H
I	J	K	L

- a) letras[2,2] - letras[2,4] - letras[3,2]
- b) letras[1,1] - letras[1,3] - letras[2,1]
- c) letras[3,3] - letras[3,5] - letras[4,3]
- d) letras[1,1] - letras[3,1] - letras[1,2]
- e) letras[1,1] - letras[1,3] - letras[2,1]

5. Usando o mesmo array do exercício anterior, quais letras estão nas posições `letras[1,2]`, `letras[2,0]` e `letras[0,2]`?

- a) J, C, I
- b) E, G, H
- c) G, I, C
- d) I, G, D
- e) G, I, D

6

Arrays

Mãos à obra!

Carlos M. R. Souza
77.357.90001-93

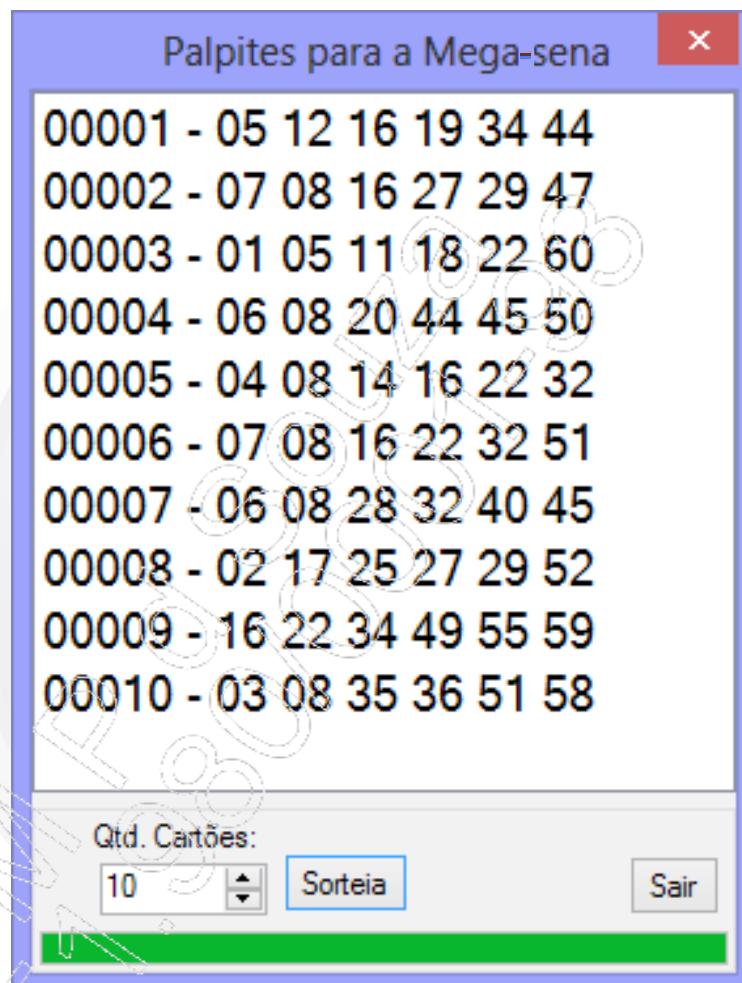


IMPACTA
EDITORA

Laboratório 1

A - Criando um projeto que gera cartões da mega-sena

1. Abra o projeto que está na pasta **Cap_06\5_Sena_Lab1** fornecida pelo instrutor. Ele deve gerar cartões da mega-sena contendo números aleatórios no intervalo de 1 a 60;



2. Para gerar números aleatórios, primeiro declare uma variável com abrangência para toda a classe:

```
Random rn = new Random();
```

3. Para gerar números no intervalo de 1 até 60, utilize este código:

```
int dezena = rn.Next(1, 61);
```

4. Utilize o evento Click do botão btnSair:

```
private void btnSair_Click(object sender, EventArgs e)
{
    this.Close();
}
```

5. Utilize o evento Click do botão btnSorteia:

```
private void btnSorteia_Click(object sender, EventArgs e)
{
    // declarar variável para armazenar a quantidade
    // total de cartões

    // limpar os itens do ListBox

    // zerar a propriedade Value da ProgressBar

    // configurar o valor máximo da ProgressBar
    // para o total de cartões

    // loop para gerar cada um dos cartões, deve contar de
    // um até o total de cartões

    // declarar array para armazenar as 6 dezenas
    // de 1 cartão

    // declarar variável para o contador de dezenas

    // enquanto o contador de dezenas for menor que 6

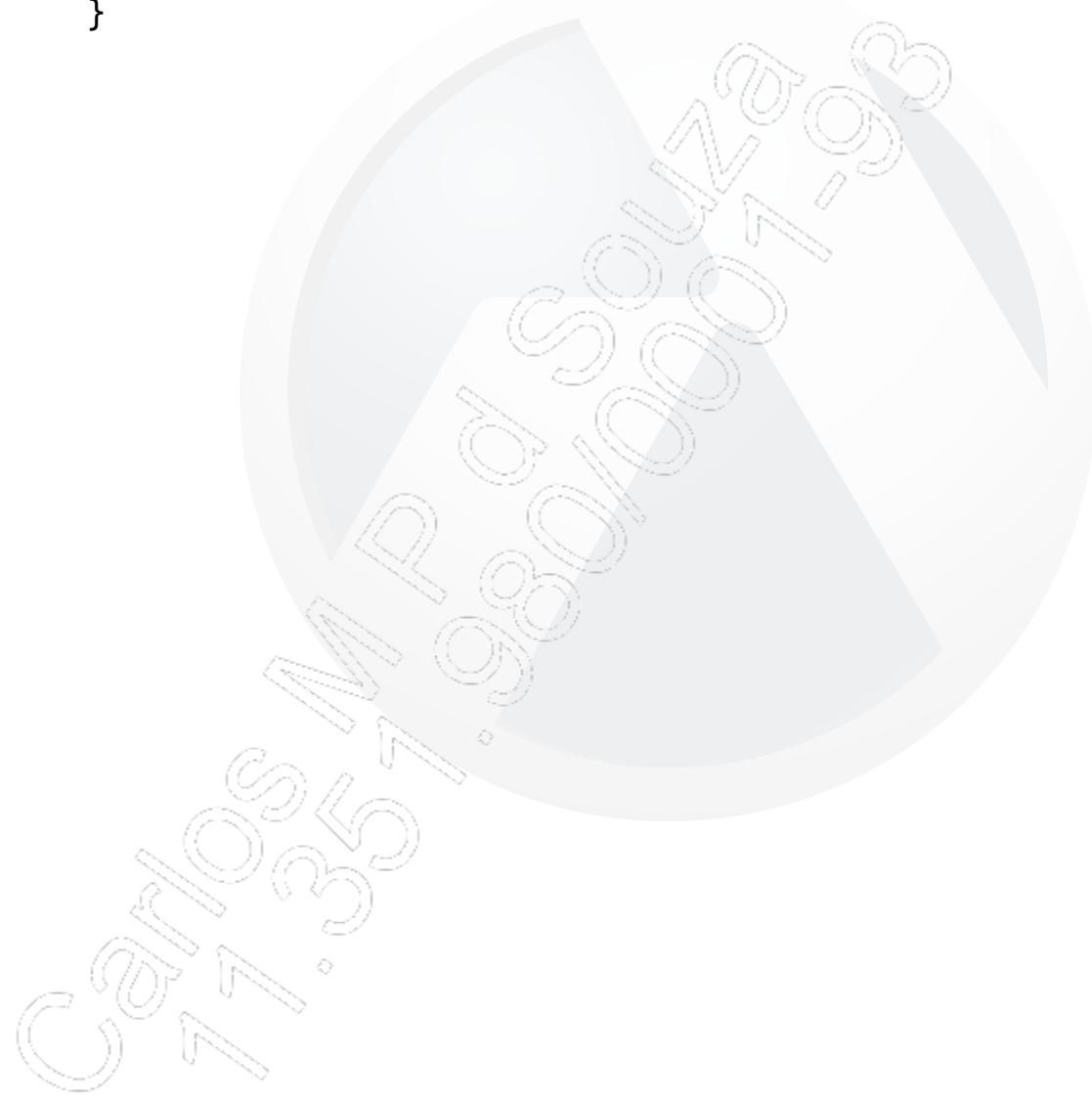
    // declarar variável para armazenar um número
    // aleatório de 1 a 60

    // se o número sorteado já existir no array de
    // dezenas, voltar para o topo deste loop

    // adicionar a dezena sorteada no array
    // de dezenas e incrementar o contador de dezenas
}
```

C# - Módulo I

```
// ordenar o array de dezenas  
  
// declarar variável string para concatenar as  
// 6 dezenas do array  
  
// loop para percorrer o array e concatenar as dezenas  
  
// adicionar no ListBox o string contendo as 6 dezenas  
  
// incrementar a propriedade Value da ProgressBar  
  
}
```



Manipulando arquivos texto

7

- ✓ Classes StreamWriter e StreamReader;
- ✓ Exemplo.

Carlos M. B. Souza
77.357.90007-93



IMPACTA
EDITORA

7.1. Classes StreamWriter e StreamReader

Neste capítulo, vamos conhecer as classes **StreamWriter** e **StreamReader**.

7.1.1. StreamWriter

Esta classe é responsável pela gravação de arquivos no formato texto. Seus principais construtores são:

- **public StreamWriter(string fileName)**

Cria um objeto **StreamWriter** direcionado ao arquivo identificado por **fileName**.

- **public StreamWriter(string fileName, Encoding encoding)**

Cria um objeto **StreamWriter** direcionado ao arquivo identificado por **fileName**. Encoding define a forma como os caracteres serão gravados, por exemplo:

- **Encoding.Default**: Padrão ANSI, cada caractere ocupa 1 byte;
- **Encoding.Unicode**: Padrão UNICODE, cada caractere ocupa 2 bytes.

- **public StreamWriter(string fileName, bool append, Encoding encoding)**

Cria um objeto **StreamWriter** direcionado ao arquivo identificado por **fileName**. Se **append** for **true**, grava a partir do final do arquivo; se for **false**, “sobregrava” o arquivo.

Os principais métodos da classe **StreamWriter** são:

- **void Write(string)**: Grava todo o string no arquivo texto;
- **void WriteLine(string)**: Grava uma linha do arquivo inserindo no final um caractere de fim de linha, normalmente 0x0D seguido de um 0x0A (“\r\n”);
- **void Close()**: Fecha o manipulador de arquivo criado na memória.

7.1.2. StreamReader

Classe utilizada para efetuar a leitura de arquivos texto. Seus principais construtores são:

- **public StreamReader(string fileName)**

Cria um objeto **StreamReader** direcionado ao arquivo identificado por **fileName**.

- **public StreamReader(string fileName, Encoding encoding)**

Cria um objeto **StreamReader** direcionado ao arquivo identificado por **fileName**. **Encoding** define a forma como os caracteres estão gravados.

A propriedade principal da classe **StreamReader** é:

- **bool EndOfStream**: Sinaliza se já lemos a última linha do arquivo.

Os principais métodos dessa classe são:

- **string ReadToEnd()**: Lê todo o conteúdo do arquivo e retorna com o string correspondente;
- **string ReadLine()**: Lê uma linha do arquivo e avança para a próxima linha. Retorna com o string correspondente à linha que foi lida;
- **int Read()**: Lê o próximo caractere do arquivo e retorna o código do caractere lido ou então -1 se for fim do arquivo;
- **void Close()**: Fecha o manipulador de arquivo criado na memória.

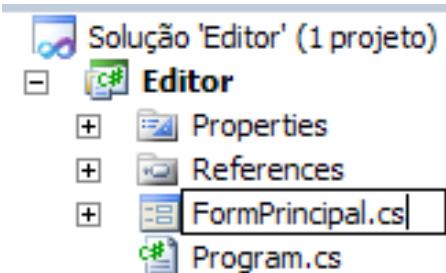
7.2. Exemplo

Neste exemplo, vamos criar um editor de textos semelhante ao bloco de notas do Windows:

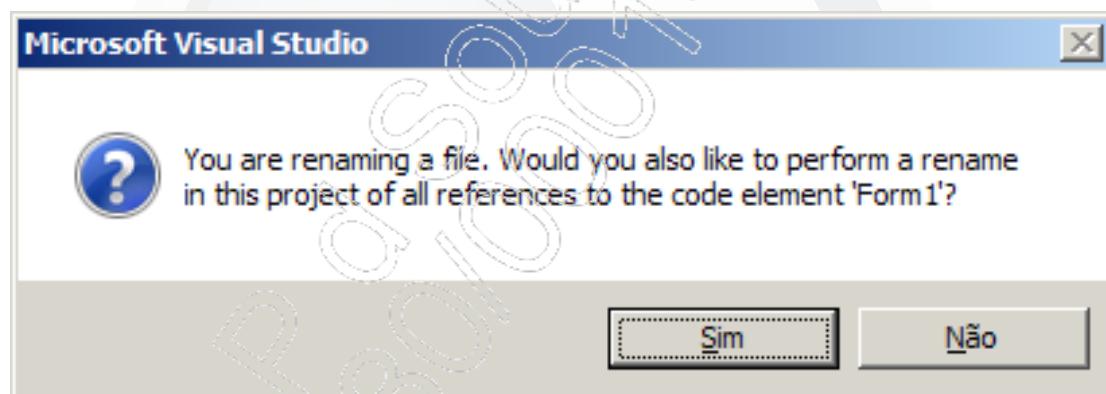
1. Crie um novo projeto Windows Form chamado **Editor**;

C# - Módulo I

2. Depois de criado o projeto, selecione o **Solution Explorer** e altere o nome do formulário de **Form1** para **FormPrincipal**:



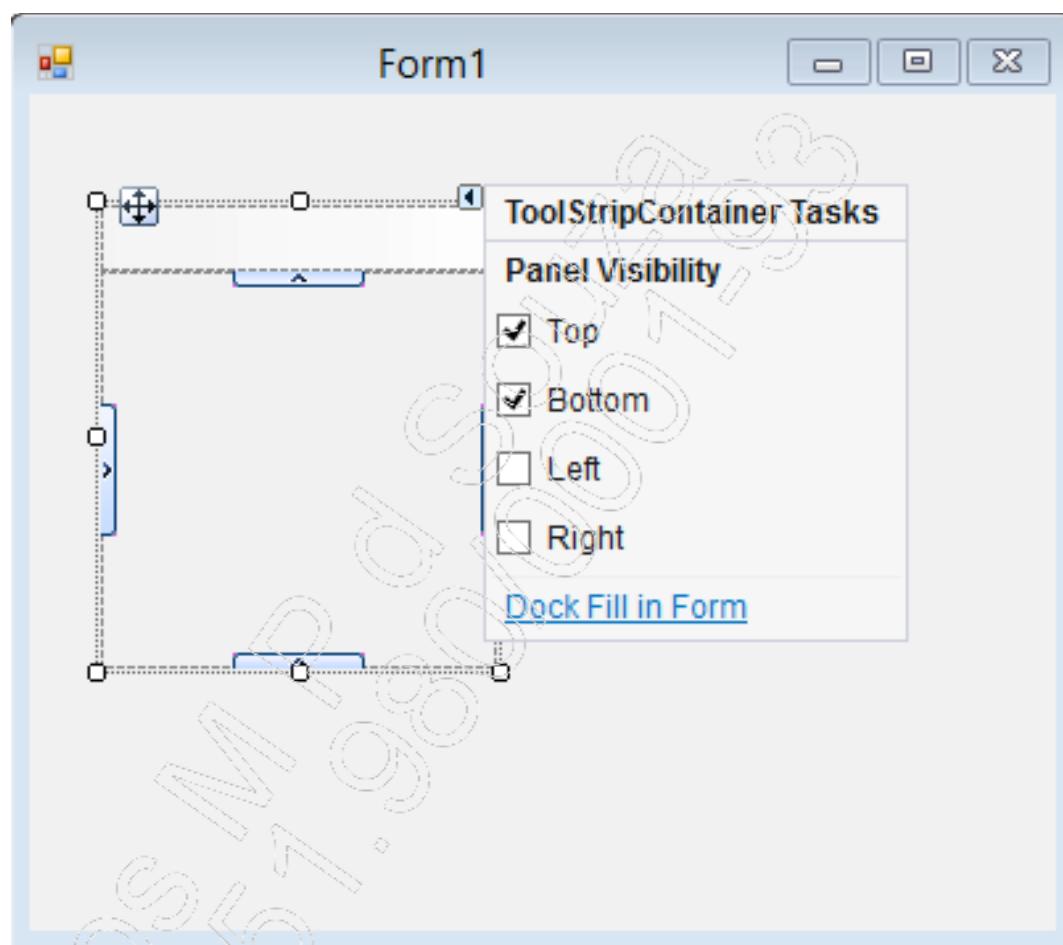
3. Assim que confirmar a alteração do nome, o VS2012 mostrará a janela a seguir. Clique em **Sim** e o nome da classe **Form1** também será alterado para **FormPrincipal**:

A screenshot of the Microsoft Visual Studio code editor. The code is written in C# and shows a class definition. The class is named "FormPrincipal" and is derived from the "Form" class. The code includes the constructor "public FormPrincipal()", the call to "InitializeComponent()", and the closing brace for the class definition.

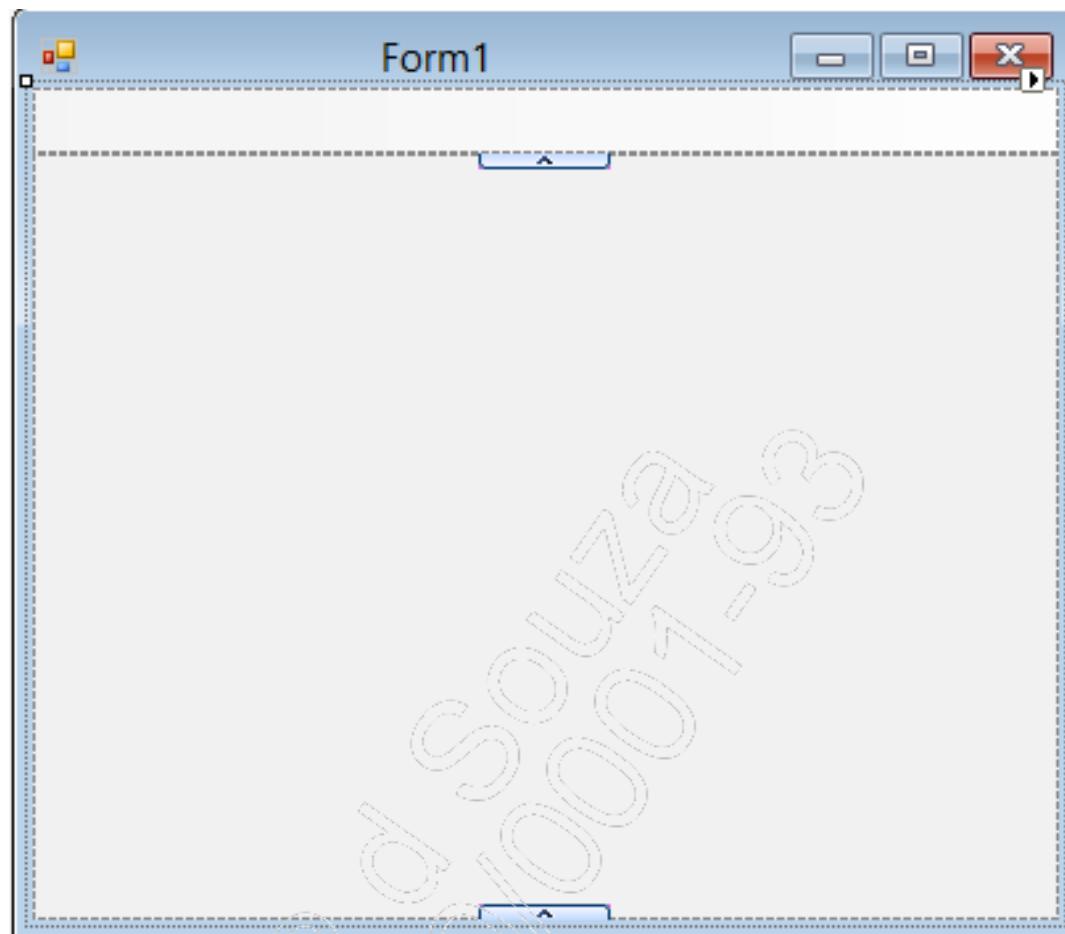
4. Altere as seguintes propriedades do formulário:

- **StartPosition:** CenterScreen;
- **Text:** Editor de Textos.

5. Coloque sobre o form um componente **ToolStripContainer**. Ele é utilizado como container para menus, barras de ferramentas e barras de status;



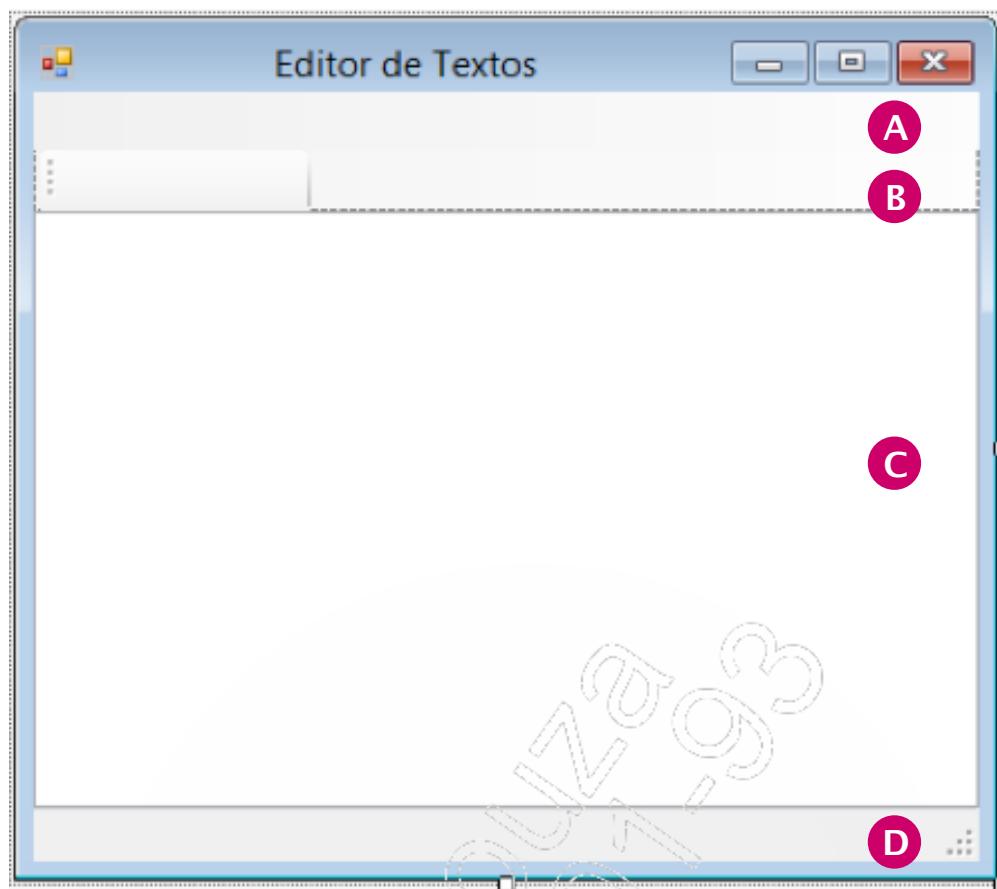
6. Selecione **Dock Fill in Form** para que ele ocupe toda a área do formulário. Desabilite os painéis laterais, da esquerda e da direita;



7. Traga para dentro do **ToolStripContainer** os componentes da imagem a seguir:

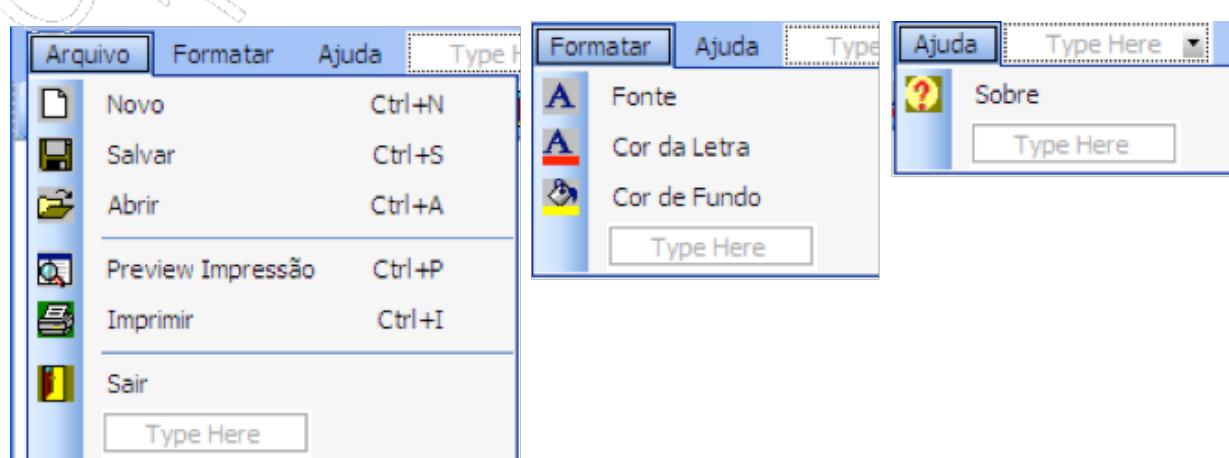
Observações:

- Abra o painel superior do **ToolStripContainer** antes de trazer os componentes A e B;
- Abra o painel inferior do **ToolStripContainer** antes de trazer o componente D.



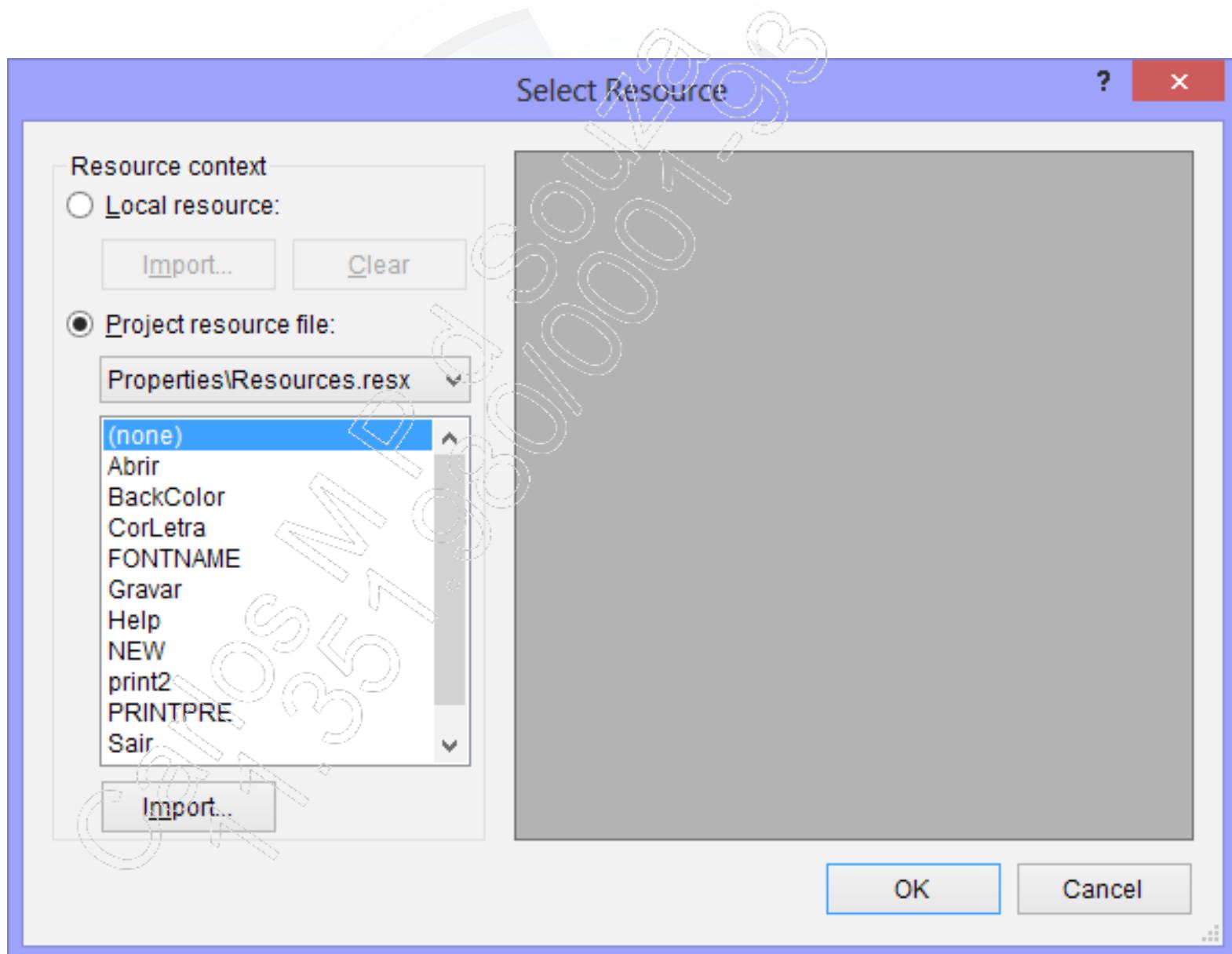
- **A - MenuStrip:** Para montar o menu principal;
- **B - ToolStrip:** Para criar uma barra de ferramentas;
- **C - TextBox:** Para armazenar o texto do editor. Altere as propriedades:
 - **MultiLine:** true;
 - **Dock:** Fill (toda a tela);
 - **Name:** tbxEditor;
 - **ScrollBars:** Vertical.
- **D - StatusStrip:** Barra de status para sinalizar informações sobre o editor.

8. Altere o **MenuStrip** de acordo com a imagem a seguir:



- Digite o título da opção diretamente sobre ela. Não digite a combinação de teclas de atalho para a opção;
- Utilize a propriedade **ShortcutKeys** para selecionar a tecla de atalho que aparece ao lado do título da opção;
- Utilize a propriedade **Image** para selecionar a imagem que será utilizada para cada uma delas.

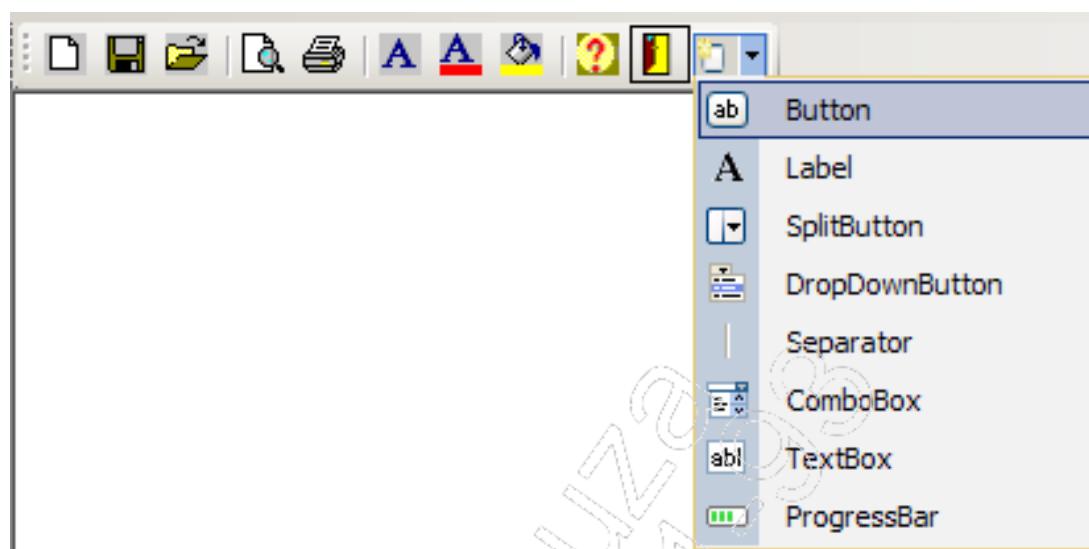
9. Mantenha selecionada a opção **Project resource file** para que as mesmas imagens possam ser utilizadas na barra de ferramentas;



- Os nomes das imagens para o menu são os mostrados na imagem anterior e estão disponíveis na pasta **1_Imagens\Toolbar**, fornecida pelo instrutor;

- Para criar um separador (linha separando grupos de opções), utilize um hífen (-).

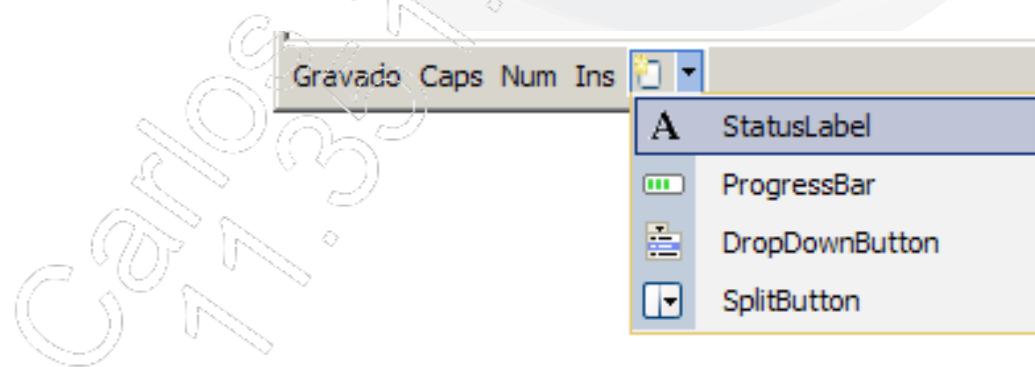
10. Altere o componente **ToolStrip** adicionando os botões ilustrados na imagem a seguir:



- Utilize o **seletor (A)** para incluir cada botão na ToolBar;
- Utilize a propriedade **Image** para selecionar a imagem. As mesmas imagens adicionadas no menu deverão estar disponíveis.

11. Configure o componente **StatusStrip**:

- Utilizando o seletor, inclua 4 **StatusLabel** na **StatusStrip**;



- Altere a propriedade **Name** de cada um deles para **IblStatus**, **IblCaps**, **IblNum** e **IblIns**.

12. A barra de status deve mostrar os status das teclas CapsLock, NumLock e Insert, porém, diferentemente das teclas CapsLock e NumLock, que mantêm o mesmo status em todos os aplicativos abertos, isso não ocorre com a tecla Insert. O Windows não gerencia o status dessa tecla. Precisaremos gerenciar isso dentro da aplicação da seguinte forma:

C# - Módulo I

```
namespace Editor
{
    public partial class FormPrincipal : Form
    {
        // como o Windows não gerencia o status do Insert,
        // criaremos uma variável para isso
        bool insertStatus = true;
        ...
    }
}
```

13. Utilize os seguintes métodos auxiliares para testar e mostrar o status das teclas na **StatusStrip**:

```
...
    bool insertStatus = true;

    public FormPrincipal()
    {
        InitializeComponent();
    }

    // omitindo o modificador de acesso o default é private
    /* private */ void verificaTeclaCaps()
    {
        if (Console.CapsLock)
            lblCaps.ForeColor = Color.Blue;
        else
            lblCaps.ForeColor = Color.Gray;
    }

    void verificaTeclaNum()
    {
        if (Console.NumberLock)
            lblNum.ForeColor = Color.Blue;
        else
            lblNum.ForeColor = Color.Gray;
    }

    void verificaTeclaIns()
    {
        // a tecla Ins não é controlada pelo Windows
        // por isso criamos a variável insertStatus
        if (insertStatus)
            lblIns.ForeColor = Color.Blue;
        else
            lblIns.ForeColor = Color.Gray;
    }
}
```

14. No evento **Load** do formulário, execute estes três métodos para mostrar o status das teclas quando o formulário for carregado:

```
private void FormPrincipal_Load(object sender, EventArgs e)
{
    verificaTeclaCaps();
    verificaTeclaIns();
    verificaTeclaNum();
}
```

15. Perceba que o status das teclas só é verificado quando carregamos o formulário, depois disso não adianta pressionarmos Caps, Num ou Insert que a tela não muda. É necessário fazer um evento **KeyDown** para o formulário, mas, antes disso, altere a propriedade **KeyPreview** do formulário para **true**:

```
/*
 * Para que o evento KeyDown do formulário seja
 * executado independente de qual controle tem foco,
 * precisamos alterar a propriedade KeyPreview para true
 */
private void FormPrincipal_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.CapsLock)
        verificaTeclaCaps();

    if (e.KeyCode == Keys.NumLock)
        verificaTeclaNum();

    if (e.KeyCode == Keys.Insert)
    {
        // o Windows não controla status do insert,
        // temos que fazer isso via programação
        insertStatus = !insertStatus;
        verificaTeclaIns();
    }
}
```

C# - Módulo I

16. Como o Windows não gerencia a tecla Insert, veremos que, no TextBox, o Insert está sempre ligado, ou seja, sempre que digitarmos, o texto à direita do cursor será empurrado pra frente, ele nunca será sobreescrito – a não ser que selecionemos parte do texto. Vamos criar o recurso de sobreescrita usando o evento **KeyPress** do TextBox. Lembre-se de que o evento KeyPress é disparado por teclas de texto e ANTES da tecla ser inserida no TextBox;

```
private void tbxEditor_KeyPress(object sender, KeyPressEventArgs e)
{
    // se o insert estiver desligado E
    if (!insertStatus &&
        // a tecla pressionada não for BackSpace E
        e.KeyChar != '\x8' &&
        // não existir nada selecionado
        tbxEditor.SelectionLength == 0)
    {
        // selecionar 1 caractere à direita do cursor
        tbxEditor.SelectionLength = 1;
    }
}
```

17. Para sinalizar na **StatusStrip** que o texto foi alterado, usaremos o evento **TextChanged** de **tbxEditor**.

```
// Evento TextChanged de tbxEditor é executado toda vez
// que o texto contido nele for alterado
private void tbxEditor_TextChanged(object sender, EventArgs e)
{
    lblStatus.Text = "Modificado";
}
```

18. Traga para o formulário um componente **SaveFileDialog** para podermos fazer a opção **Salvar** do menu. Altere as seguintes propriedades:

- **DefaultExt**: txt;
- **FileName**: Documento1.txt;
- **Filter**: Arquivos Texto(*.txt)|*.txt|Todos os arquivos(*.*)|*.*.

```
private void salvarToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        // objeto para salvar arquivo texto
        // cria uma referência na memória (manipulador de
        // arquivo) associada ao arquivo
        StreamWriter sw = new StreamWriter(
            saveFileDialog1.FileName);
        // grava todo o texto (na memória ainda)
        sw.Write(tbxEditor.Text);
        // fecha o manipulador de arquivo na memória
        // É agora que realmente grava no arquivo
        sw.Close();
        // libera a memória usada pelo StreamWriter
        sw.Dispose();
        // Faz o coletor de lixo (GC) liberar tudo que
        // não está sendo usado
        // GC.Collect();
        lblStatus.Text = "Gravado";
        tbxEditor.Modified = false;
    }
}
```

19. Traga para o formulário um componente **OpenFileDialog** para podermos fazer a opção **Abrir** do menu. Altere as seguintes propriedades:

- **DefaultExt**: txt;
- **Filter**: Arquivos Texto(*.txt)|*.txt|Todos os arquivos(*.*)|*.*.

```
private void abrirToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        // objeto para ler conteúdo e arquivo texto
        StreamReader sr = new StreamReader(
            openFileDialog1.FileName);
        // lê todo o conteúdo do arquivo e armazena em
```

```
// tbxEditor
tbxEditor.Text = sr.ReadToEnd();
// fecha o manipulador de arquivo
sr.Close();
// libera memória
sr.Dispose();
// sinaliza que o texto não foi alterado
lblStatus.Text = "Gravado";
tbxEditor.Modified = false;
}
}
```

20. Agora crie um método para o evento **Click** da opção **Arquivo / Novo** do menu:

```
private void novoToolStripMenuItem_Click(object sender, EventArgs e)
{
    tbxEditor.Clear();
    saveFileDialog1.FileName = "Documento1.txt";
    lblStatus.Text = "Gravado";
    tbxEditor.Modified = false;
}
```

21. Observe que o Editor não nos avisa que o texto não foi salvo quando abrimos um novo ou quando fechamos o aplicativo. Para isso, crie um método auxiliar que deve ser executado em 3 situações diferentes:

```
/*
 * Se o texto foi modificado, vai perguntar
 * ao usuário se deseja perder alterações.
 * Em caso afirmativo, retorna true; caso contrário,
 * retorna false.
*/
bool perderAlteracoes()
{
    // se o texto não foi modificado, não há o que perder
    // pode prosseguir na operação de Abrir, Sair ou Novo
    if (!tbxEditor.Modified) return true;
    // se chegar aqui é porque o texto foi alterado.
    // Perguntar se quer realmente perder as alterações
}
```

```

    if (MessageBox.Show(
        "Texto não foi salvo, perder alterações?",
        "Cuidado!!!", // título da janela
        MessageBoxButtons.YesNo, // conjunto de botões
        MessageBoxIcon.Warning, // ícone ao lado da mensagem
        MessageBoxDefaultButton.Button2 // botão com foco
            ) == DialogResult.Yes)
        return true;
    else
        return false;
}

```

22. Agora altere os métodos das opções Arquivo / Abrir e Arquivo / Novo:

```

private void abrirToolStripMenuItem_Click(object sender, EventArgs
e)
{
    if (!perderAlteracoes()) return;
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        ...
    }
}

private void novoToolStripMenuItem_Click(object sender, EventArgs
e)
{
    if (!perderAlteracoes()) return;
    tbxEditor.Clear();
    ...
}

```

23. Crie um método para o evento **FormClosing** do formulário. Este método será executado quando mandarmos fechar o formulário, por qualquer meio, mas antes dele ser fechado:

```

private void FormPrincipal_FormClosing(object sender, FormClosingEventArgs
e)
{
    if (!perderAlteracoes())
    {
        // cancela o fechamento do formulário
        e.Cancel = true;
    }
}

```

C# - Módulo I

24. Traga para o formulário um componente **FontDialog** e um **ColorDialog**. Crie os métodos das opções **Formatar**:

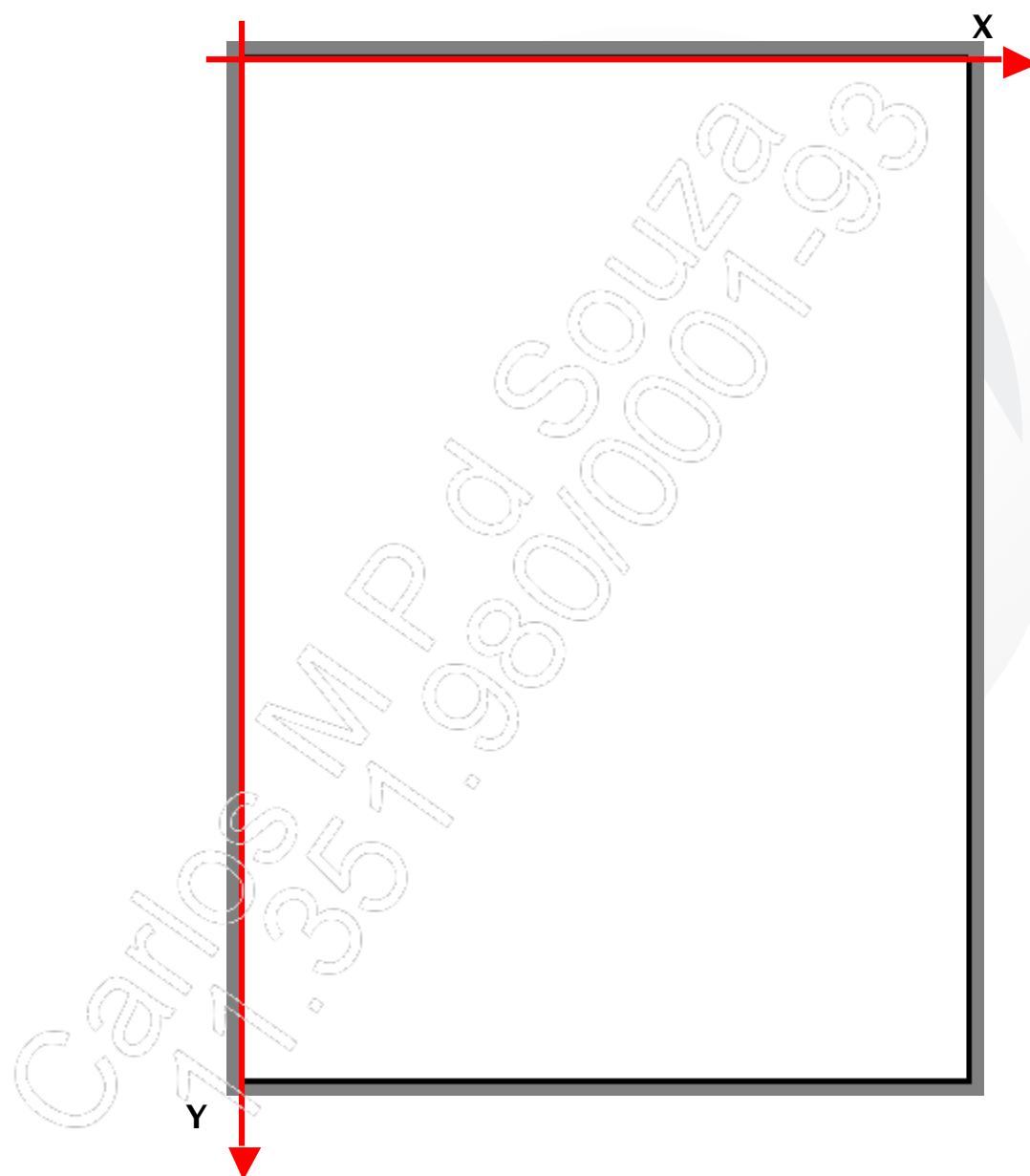
```
private void fonteToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (fontDialog1.ShowDialog() == DialogResult.OK)
        tbxEditor.Font = fontDialog1.Font;
}

private void corLetraToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (colorDialog1.ShowDialog() == DialogResult.OK)
        tbxEditor.ForeColor = colorDialog1.Color;
}

private void corFundoToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (colorDialog1.ShowDialog() == DialogResult.OK)
        tbxEditor.BackColor = colorDialog1.Color;
}
```

25. Para criar as opções de impressão, você precisará dos seguintes componentes:

- **PrintDocument**: Responsável pela impressão;
- **PrintPreviewDialog**: Janela para exibir a impressão na tela;
- **PrintDialog**: Janela que aparece no momento da impressão para que o usuário possa selecionar a impressora e definir as características da impressão.



26. Os componentes **PrintPreviewDialog** e **PrintDialog** precisam estar ligados ao **PrintDocument** através da propriedade **Document**. Para o componente **PrintDocument**, crie um método para o evento **PrintPage**. Este método será executado em loop pelo **PrintDocument** até que retorne não haver mais páginas para imprimir;

C# - Módulo I

```
// contador de linhas e contador de página
int ctLinhas, ctPag;

private void printDocument1_PrintPage(object sender, System.Drawing.
Printing.PrintPageEventArgs e)
{
    //Bitmap bmp = new Bitmap(@"C:\...\IMAGEM");
    Bitmap fundo = new Bitmap("fundo.bmp");
    // variável para posição da linha ( Y )
    int yPos = 20;
    // cor da letra
    // o preenchimento das letras pode ser sólido
    Brush br = Brushes.Black;
    // o preenchimento das letras pode ser feito com uma figura
    //Brush br = new TextureBrush(bmp);
    // pode ser feito com linhas cruzadas
    //Brush br = new HatchBrush(HatchStyle.Cross, Color.Blue);
    // pode ser feito com linhas diagonais
    //Brush br = new
    //HatchBrush(HatchStyle.DashedDownwardDiagonal,
    //Color.White, Color.Blue);
    // pode ser feito com gradiente de cores
    //Brush br = new LinearGradientBrush(new Point(0, 0),
    //new Point(0, 100), Color.Blue, Color.Aqua);
    // unidade de medida para X e Y em milímetros
    e.Graphics.GraphicsUnit = GraphicsUnit.Millimeter;
    // desenha uma imagem no fundo da folha
    e.Graphics.DrawImage(fundo, 15, 10, 190, 270);
    // desenha um retângulo na borda da folha
    e.Graphics.DrawRectangle(new Pen(Color.Black, 1), 15, 10, 190, 270);
    // imprime o número da página
    e.Graphics.DrawString("Pag.: " + (ctPag++),
        new Font("Arial", 12, FontStyle.Bold), br, 185, 12);

    // medir o tamanho da letra de acordo com a fonte do texto
    SizeF tLetra = e.Graphics.MeasureString("X",
        tbxEditor.Font);
```

```
// enquanto não for fim de página e não for fim do texto
while (yPos < 250 && ctLinhas < tbxEditor.Lines.Length)
{
    // imprimir a linha do texto
    e.Graphics.DrawString(tbxEditor.Lines[ctLinhas++],
        tbxEditor.Font,
        br, 20, yPos);
    // incr. posição Y de acordo com a altura da letra
    yPos += (int)tLetra.Height + 2;
}
// se true, salta para próxima página e volta para
// este método para imprimir nova página
// se false, finaliza a impressão
e.HasMorePages = ctLinhas < tbxEditor.Lines.Length;
}
```

27. Utilize o evento Click da opção Arquivo / Preview Impressão;

```
private void previewDeImpreToolStripMenuItem_Click(object sender,
EventArgs e)
{
    ctLinhas = 0;
    ctPag = 1;

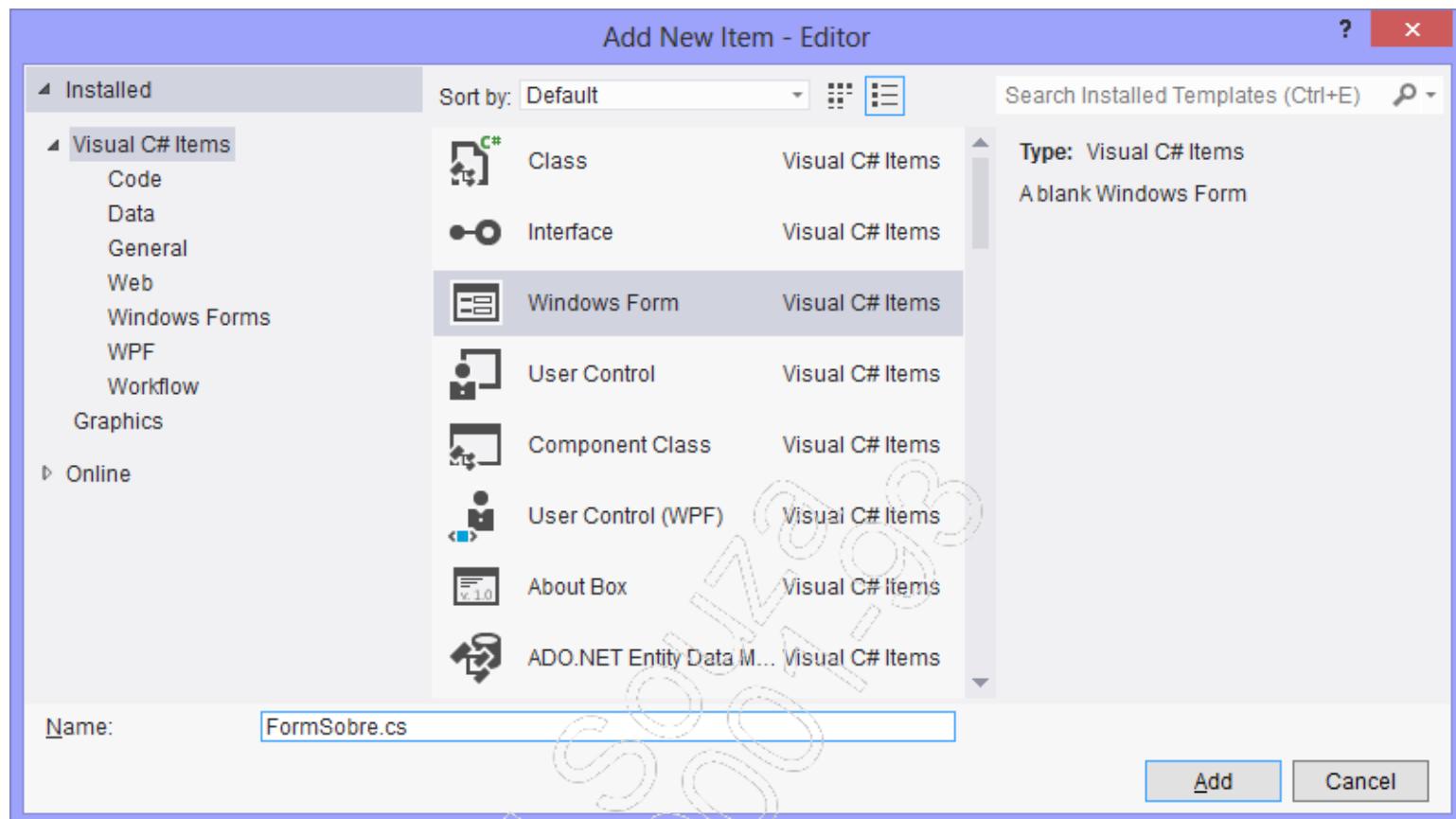
    printPreviewDialog1.WindowState =
        FormWindowState.Maximized;
    printPreviewDialog1.ShowDialog();
}
```

28. Utilize o evento Click da opção Arquivo / Imprimir;

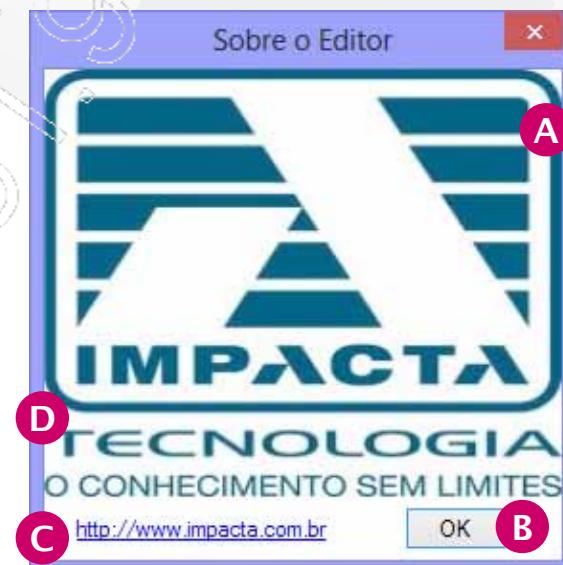
```
private void imprimirToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
        ctLinhas = 0;
        ctPag = 0;
        printDocument1.Print();
    }
}
```

C# - Módulo I

29. Adicione um novo formulário ao projeto: **Project / Add Windows Form**. Altere o nome do projeto para **FormSobre**:



30. Montar a tela de acordo com a imagem a seguir:



- **A - Panel:** Utilize a propriedade **BackgroundImage** para colocar a imagem;
- **B - Panel;**
- **C - LinkLabel;**
- **D - Label** com propriedade Color = Transparent.

31. O evento **Click** do botão **OK** deve fechar o **form**. Utilize o método **Close()**;

32. O evento **Click** do **LinkLabel** deve abrir a página da Impacta;

```
private void linkLabel1_LinkClicked(object sender,  
LinkLabelLinkClickedEventArgs e)  
{  
    // using System.Diagnostics  
    Process.Start("http://www.impacta.com.br");  
}
```

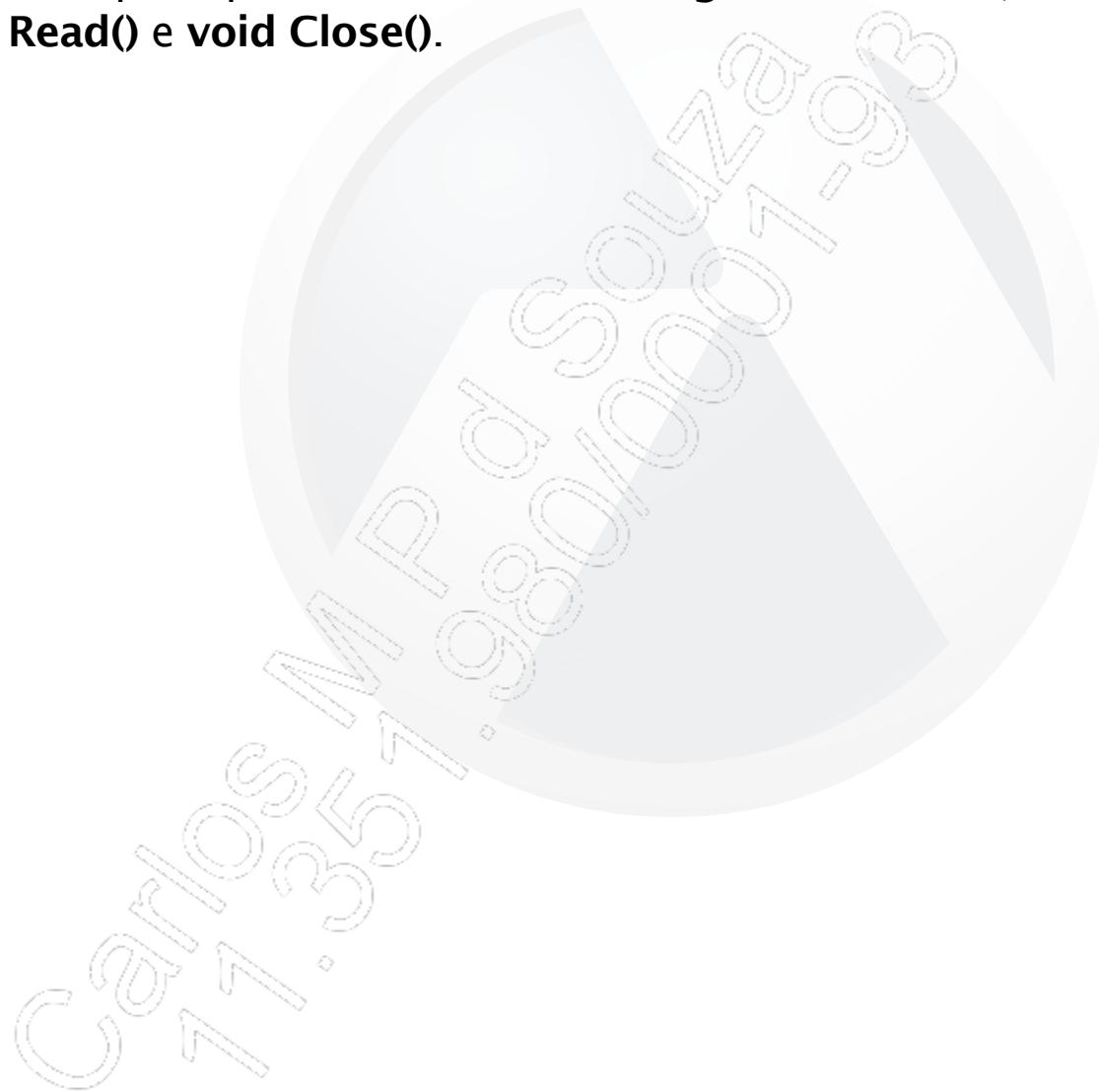
33. Utilize o evento **Click** da opção **Ajuda / Sobre** do **FormPrincipal**.

```
private void sobreToolStripMenuItem_Click(object sender, EventArgs e)  
{  
    // cria instância do formulário e exibe no vídeo  
    (new FormSobre()).ShowDialog();  
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A classe **StreamWriter** é responsável pela gravação de arquivos no formato texto. Seus principais métodos são **void Write(string)**, **void WriteLine(string)** e **void Close()**;
- A classe **StreamReader** é utilizada para efetuar a leitura de arquivos texto. Seus principais métodos são **string ReadToEnd()**, **string ReadLine()**, **int Read()** e **void Close()**.



7

Manipulando arquivos texto

Teste seus conhecimentos

CarloS M. Gómez
77.350-000
SOUZA
07-993



IMPACTA
EDITORA

1. Qual das alternativas possui os métodos corretos da classe StreamWriter?

- a) Write(string), WriteString(string)
- b) WriteText(string), Write(string)
- c) Write(string), WriteLine(string)
- d) WriteToEnd(string), WriteLine(string)
- e) WriteToEnd(string), WriteText(string)

2. Qual das alternativas possui os métodos corretos da classe StreamReader?

- a) Read(), ReadAll()
- b) ReadText(), ReadLine()
- c) Read(), ReadString()
- d) ReadLine(), ReadToEnd()
- e) ReadAll(), ReadString()

3. Qual propriedade deve ser utilizada para centralizar um formulário no vídeo?

- a) StartPosition = CenterScreen
- b) Position = CenterScreen
- c) StartPosition = ScreenCenter
- d) Position = ScreenCenter
- e) Não existe propriedade para esta finalidade

4. Qual é o evento do TextBox que pode detectar o pressionamento de qualquer tecla do teclado?

- a) KeyPress
- b) KeyDown
- c) KeyUp
- d) KeyPreview
- e) KeyHold

5. Qual é o método da classe System.Drawing.Graphics que mede o tamanho de uma string de acordo com uma determinada fonte e com a unidade de medida usada para impressão?

- a) StringLength(string, font)
- b) StringSize(string, font)
- c) MeasureString(string, font)
- d) MeasureText(string, font)
- e) StringMeasure(string, font)

Enumerações e Delegates

8

- ✓ Enumerações;
- ✓ Delegates.

Carlos M. D. Souza
77.357.9007-93



IMPACTA
EDITORA

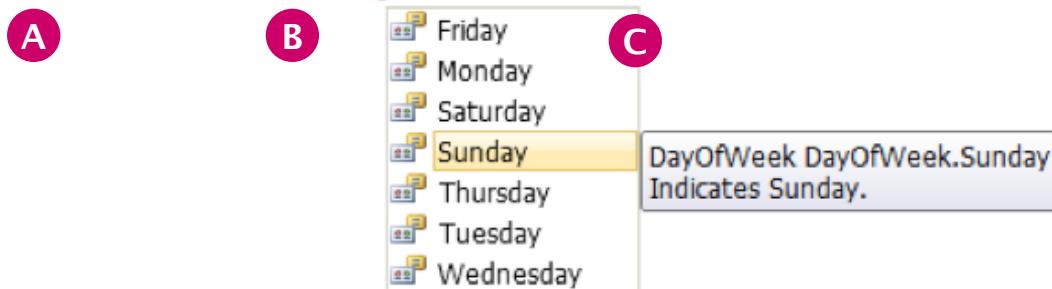
8.1. Enumerações

São um tipo de dado usado para declarar variáveis, propriedades e métodos que contêm um valor numérico. Cada um desses números possíveis recebe um nome. A plataforma .Net possui uma série de tipos enumerados. Veja o exemplo a seguir:

```
namespace System
{
    // Summary:
    //     Specifies the day of the week.
    [Serializable]
    [ComVisible(true)]
    public enum DayOfWeek
    {
        // Summary:
        //     Indicates Sunday.
        Sunday = 0,
        //
        // Summary:
        //     Indicates Monday.
        Monday = 1,
        //
        // Summary:
        //     Indicates Tuesday.
        Tuesday = 2,
        //
        // Summary:
        //     Indicates Wednesday.
        Wednesday = 3,
        //
        // Summary:
        //     Indicates Thursday.
        Thursday = 4,
        //
        // Summary:
        //     Indicates Friday.
        Friday = 5,
        //
        // Summary:
        //     Indicates Saturday.
        Saturday = 6,
    }
}
```

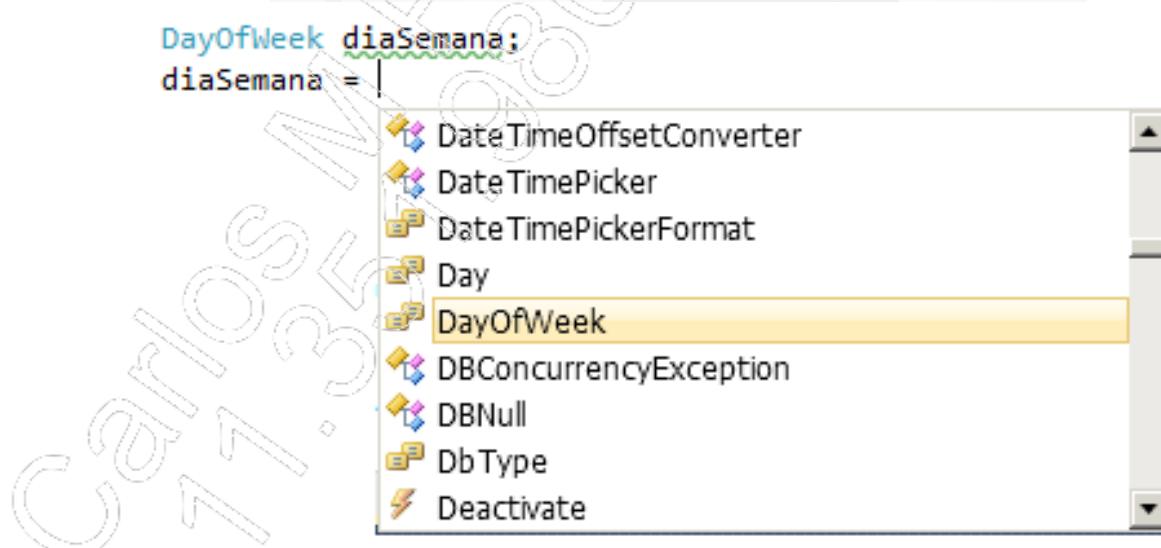
Veja que foram dados nomes aos números para representar os dias da semana.

```
DateTime dataHoje = DateTime.Now;  
if (dataHoje.DayOfWeek == DayOfWeek.Sunday)
```



- A - Propriedade de `DateTime` que retorna o dia da semana de uma data;
- B - Tipo enumerado, declarado no namespace `System`, que contém as opções possíveis para dia da semana;
- C - Opções criadas no tipo enumerado `DayOfWeek`.

Se declararmos uma variável do tipo `DayOfWeek`, ela aceitará somente uma das opções contidas no tipo enumerado.



Quando vamos atribuir valor à propriedade, o VS2012 já sugere que deve ser um dos elementos de **DayOfWeek**. Se tentarmos forçar algo diferente, um erro será detectado.

The screenshot shows a code editor with the following code:

```
struct System.Int32
Represents a 32-bit signed integer.

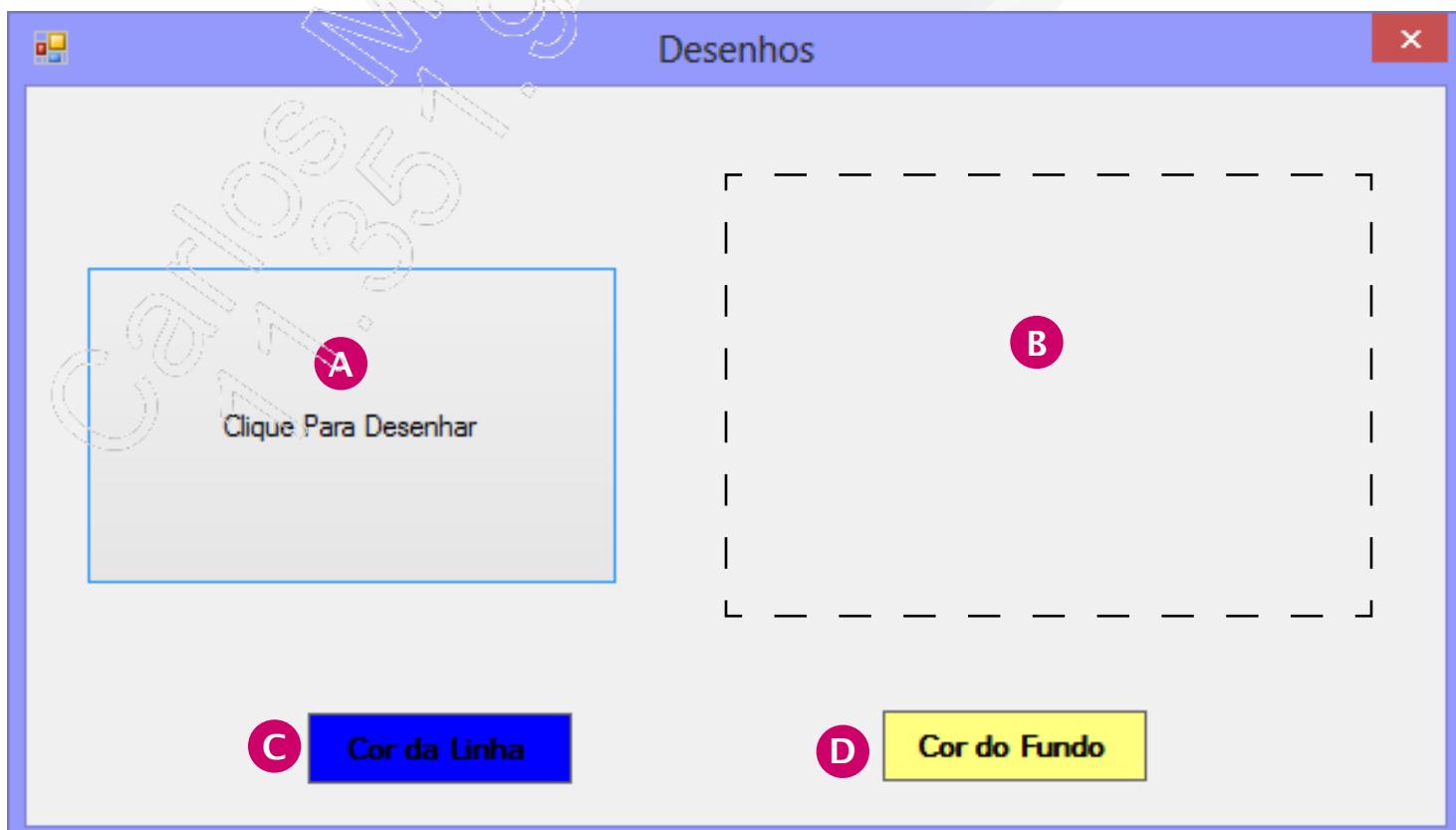
Error:
Cannot implicitly convert type 'int' to 'System.DayOfWeek'. An explicit conversion exists (are you missing a cast?)
```

A tooltip arrow points from the error message to the line `diaSemana = 2;`. Another tooltip arrow points from the message `'System.DayOfWeek' does not contain a definition for 'Domingo'` to the word `Domingo` in the line `diaSemana = DayOfWeek.Domingo;`.

8.1.1. Exemplo

Abra o projeto que está na pasta **Cap_08\1_Paint_Ex1** fornecida pelo instrutor.

Cada vez que clicarmos no botão, será feito um desenho diferente no painel à direita. Os desenhos serão: Retângulo, Elipse, Linhas Horizontal e Vertical, Linhas Diagonais, Losango e Bandeira.



1. Crie um tipo enumerado para definir as opções de desenho;

```
public partial class Form1 : Form
{
    // tipo enumerado para definir as opções do desenho que serão feitas
    enum TipoDesenho { None, Retangulo, Elipse,
        LinhasHV, LinhasDiag,
        Losango, Bandeira };
    // variável que armazena uma das opções do tipo enumerado
    // e que vai definir o desenho que será feito
    TipoDesenho tipoDes;
}
```

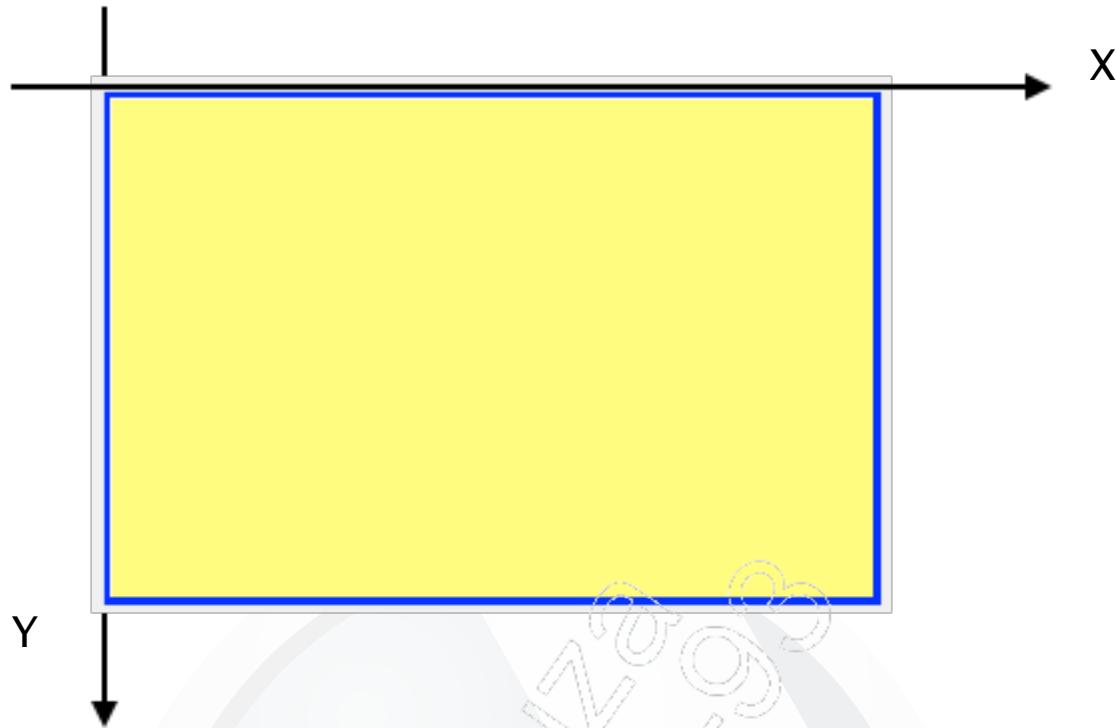
2. Crie o evento **Click** do botão que definirá o desenho;

```
private void desenhaButton_Click(object sender, EventArgs e)
{
    // verifica se a variável é menor que o último elemento
    if (tipoDes < TipoDesenho.Bandeira)
        // se for, incrementa a variável passando para o próximo desenho
        tipoDes++;
    else
        // caso contrário, volta para o desenho do retângulo
        tipoDes = TipoDesenho.Retangulo;
    // recupera o nome do tipo enumerado atual e mostra no Button
    desenhaButton.Text = Enum.GetName(typeof(TipoDesenho), (int)tipoDes);
    // força o redesenho do componente Panel que exibe a figura
    // executando então o seu evento Paint que criaremos a seguir
    pnl.Invalidate();
}
```

3. Crie o evento **Paint** do Panel. Este evento recebe a variável **PaintEventArgs** como argumento e possui todos os recursos necessários para desenhar sobre o Panel;

```
private void pnl_Paint(object sender, PaintEventArgs e)
{
    // define o preenchimento da figura, neste caso
    // sólido e da mesma cor de fundo de lblFundo (Label)
    Brush corFundo = new System.Drawing.SolidBrush(lblFundo.BackColor);
```

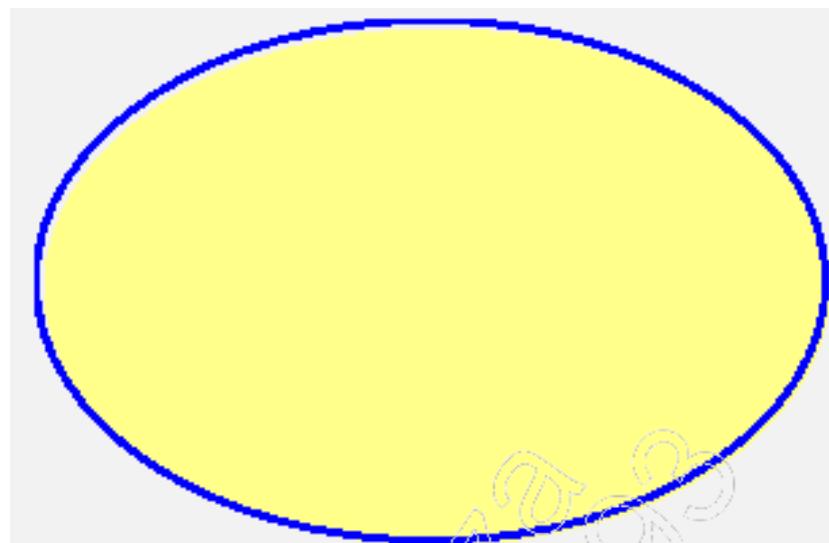
- Se estiver definido retângulo:



```
// se o desenho definido for Retângulo
if (tipoDes == TipoDesenho.Retangulo)
{
    // desenha o fundo do retângulo
    e.Graphics.FillRectangle(
        // preenchimento
        corFundo,
        // posição X e Y do canto superior esquerdo
        3, 3,
        // largura e altura do retângulo
        pnl.ClientSize.Width - 3,
        pnl.ClientSize.Height - 3);
    // desenha a linha de borda
    e.Graphics.DrawRectangle(
        // caneta com cor definida em lblLinha e espessura 3
        new Pen(lblLinha.BackColor, 3),
        // posição X e Y do canto superior esquerdo
        0, 0,
        // largura e altura do retângulo
        pnl.ClientSize.Width - 2, pnl.ClientSize.Height - 2);

}
```

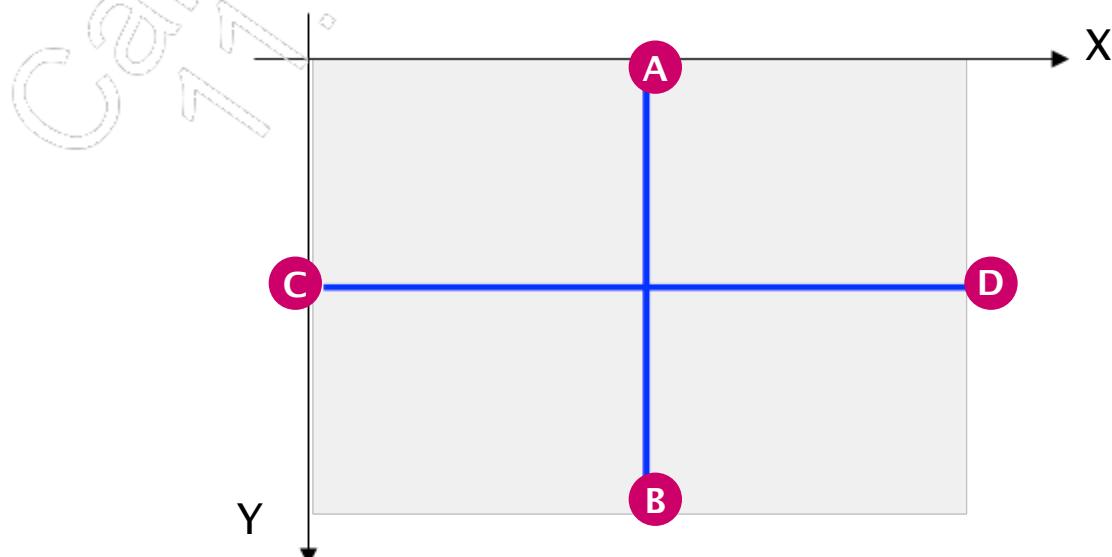
- Se for elipse, basta fornecer o retângulo em que a elipse está contida. Vai mudar apenas o nome do método:



```
// se for Elipse
else if (tipoDes == TipoDesenho.Eipse)
{
    e.Graphics.FillEllipse(corFundo, 3, 3,
        pnl.ClientSize.Width - 3, pnl.ClientSize.Height - 3);
    e.Graphics.DrawEllipse(new Pen(lblLinha.BackColor, 3), 0, 0,
        pnl.ClientSize.Width - 2,
        pnl.ClientSize.Height - 2);

}
```

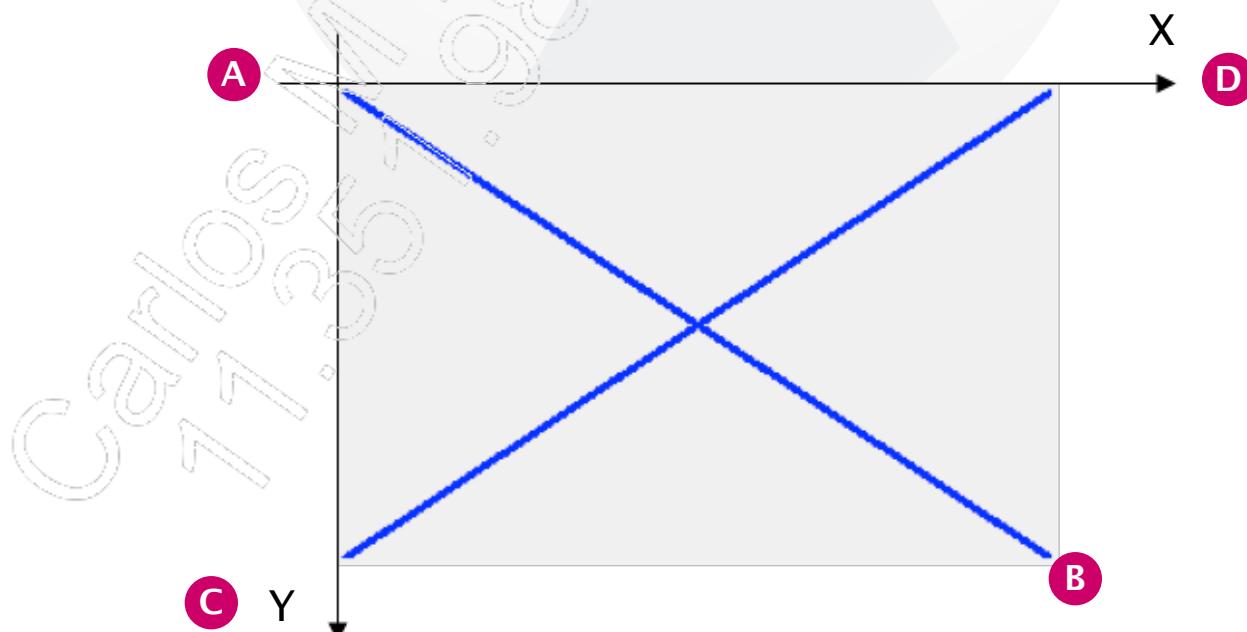
- No caso de linhas horizontal e vertical, as posições X e Y de cada ponto são:



- A - `pnl.Width / 2, 0;`
- B - `pnl.Width / 2, pnl.Height;`
- C - `0, pnl.Height / 2;`
- D - `pnl.Width, pnl.Height / 2.`

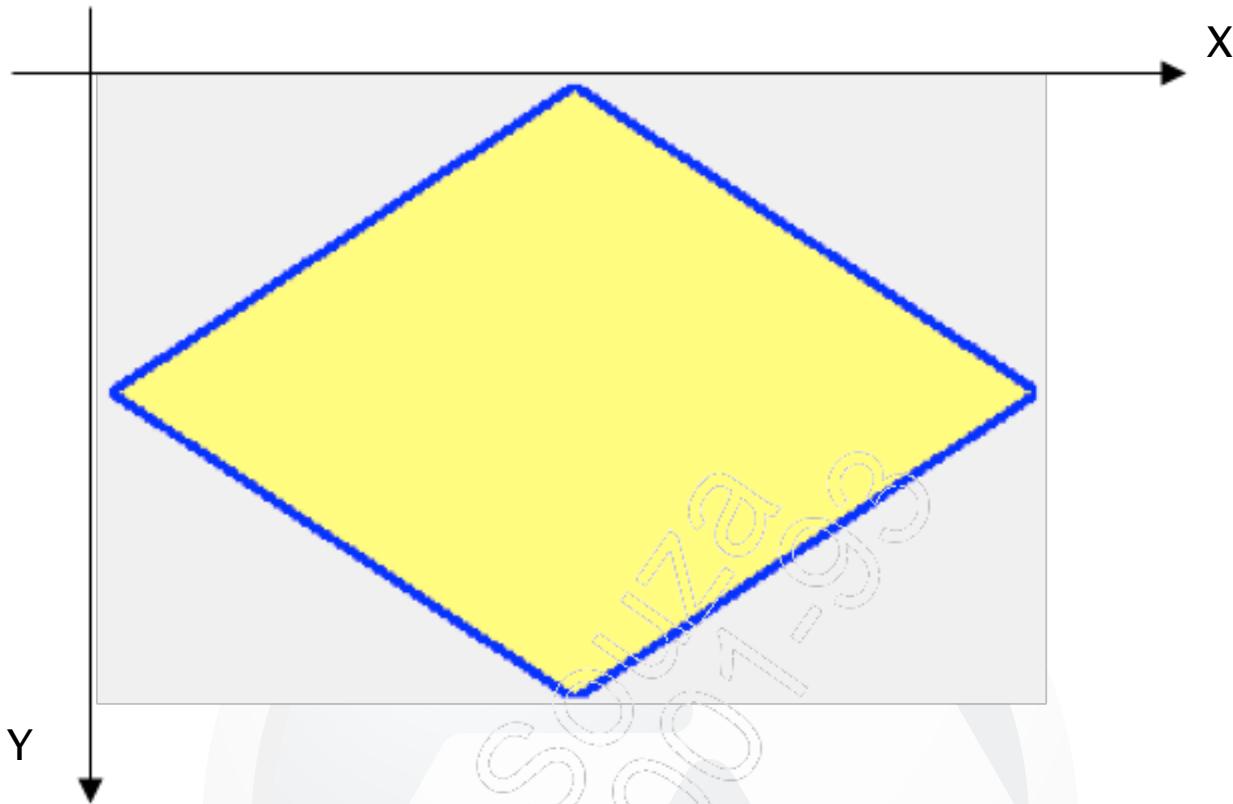
```
// linhas Horizontal e Vertical
else if (tipoDes == TipoDesenho.LinhasHV)
{
    // linha vertical no centro do panel
    e.Graphics.DrawLine(
        // caneta usada no desenho da linha
        new Pen(lblLinha.BackColor, 3),
        // posição X e Y do primeiro ponto da linha
        pnl.Width / 2, 0,
        // posição X e Y do segundo ponto da linha
        pnl.Width / 2, pnl.Height);
    // linha horizontal no centro do retângulo
    e.Graphics.DrawLine(new Pen(lblLinha.BackColor, 3), 0,
        pnl.Height / 2,
        pnl.Width, pnl.Height / 2);
}
```

- No caso de linhas diagonais, complete as lacunas a seguir:



- A. _____, _____
- B. _____, _____
- C. _____, _____
- D. _____, _____

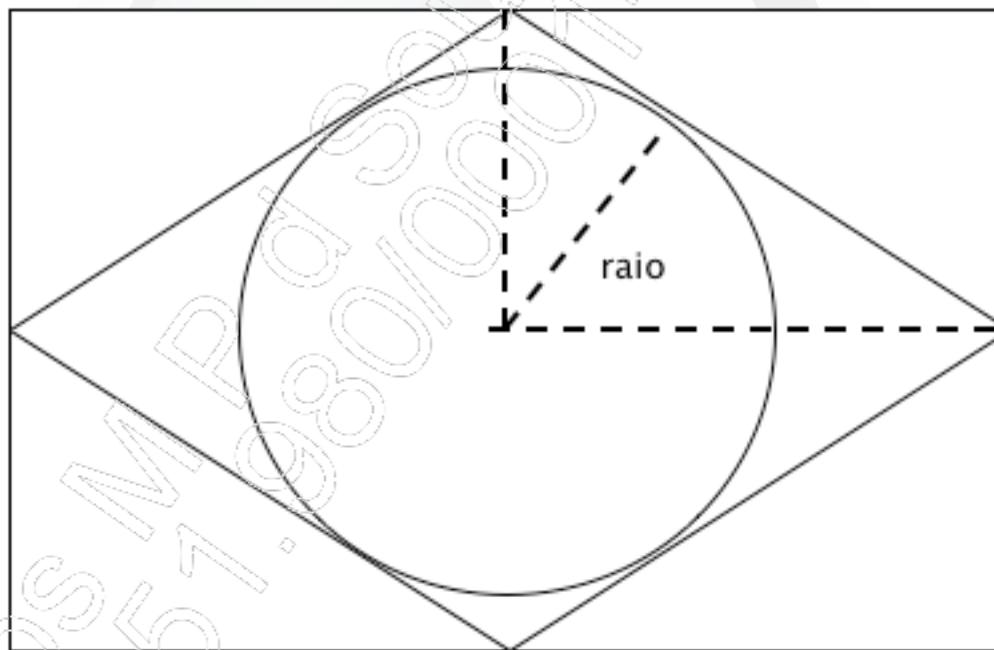
- Se for losango, utilize o código a seguir:



```
else if (tipoDes == TipoDesenho.Losango)
{
    // cria um array de pontos que serão ligados para
    // desenhar o polígono
    Point[] ptPoly = {new Point(pnl.Width/2,0),
                      new Point(pnl.Width, pnl.Height/2),
                      new Point(pnl.Width/2,pnl.Height),
                      new Point(0,pnl.Height/2)};
    // desenhar o fundo e a borda do polígono
    e.Graphics.FillPolygon(corFundo, ptPoly);
    e.Graphics.DrawPolygon(new Pen(lblLinha.BackColor, 3), ptPoly);
}
```

C# - Módulo I

- Se for bandeira, utilize o código a seguir:

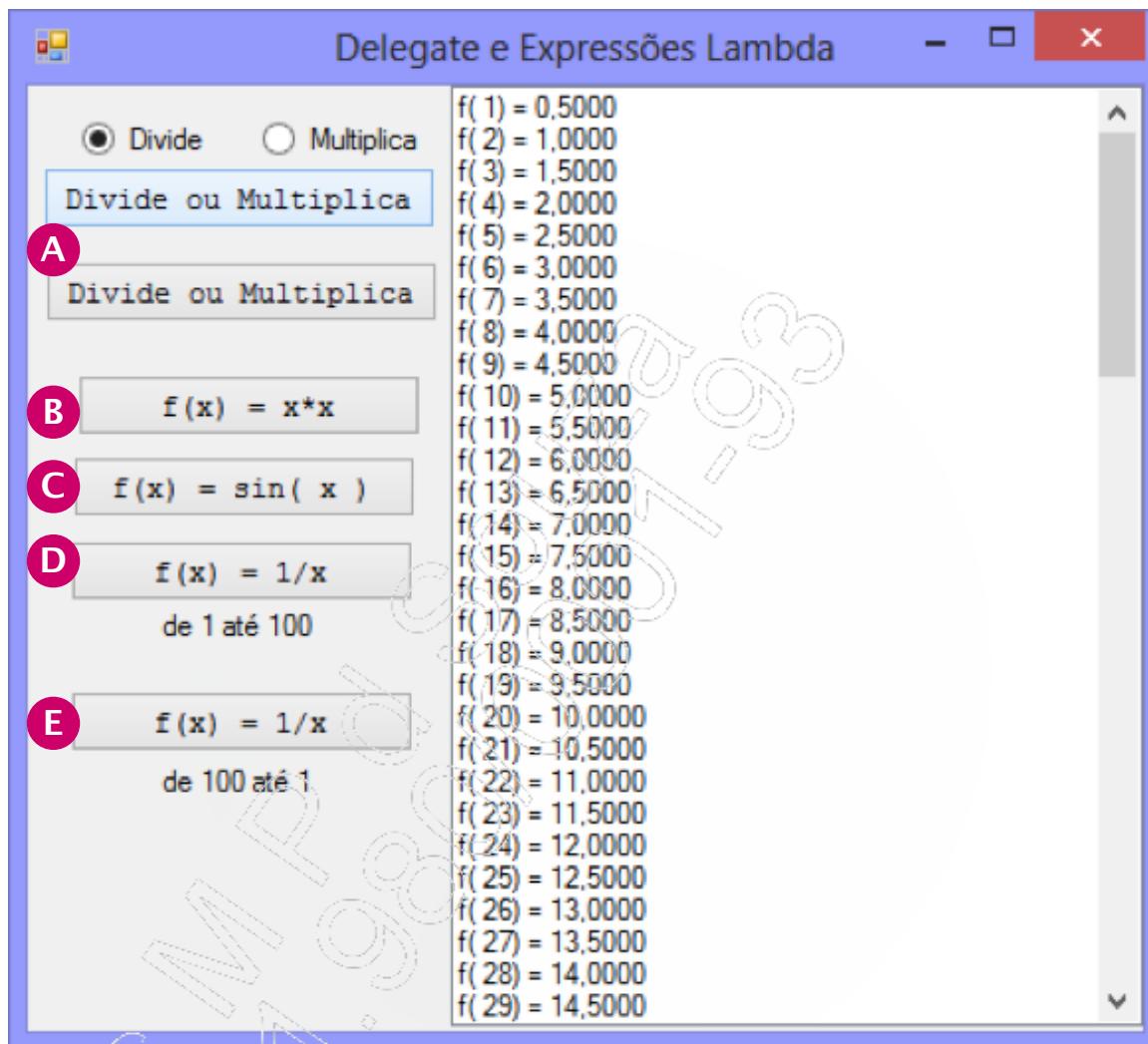


```
// bandeira do Brasil
else if (tipoDes == TipoDesenho.Bandeira)
{
    double w, h, r, b, area;
    // array de pontos para desenhar o losango da bandeira
    Point[] ptPoly = {new Point(pnl.Width/2,0),
                      new Point(pnl.Width, pnl.Height/2),
                      new Point(pnl.Width/2,pnl.Height),
                      new Point(0,pnl.Height/2)};
    // fonte para escrever ORDEM E PROGRESSO
    System.Drawing.Font drawFont = new Font("Arial", 7);
    // calcula o raio da circunferência dentro do losango
```

```
// usando a área do triângulo e o teorema de pitágoras
b = pnl.Width / 2;
w = pnl.Height / 2;
area = w * b / 2;
h = Math.Sqrt(w * w + b * b);
r = 2 * area / h - 2;
// define preenchimento verde do retângulo da bandeira
corFundo = new System.Drawing.SolidBrush(Color.Green);
// desenha o retângulo da bandeira
e.Graphics.FillRectangle(corFundo, 1, 1,
    pnl.ClientSize.Width - 1, pnl.ClientSize.Height - 1);
e.Graphics.DrawRectangle(new Pen(Color.Black, 2), 0, 0,
    pnl.ClientSize.Width - 1,
    pnl.ClientSize.Height - 1);
// define o preenchimento amarelo do losango da bandeira
corFundo = new System.Drawing.SolidBrush(Color.Yellow);
// desenha o losango
e.Graphics.FillPolygon(corFundo, ptPoly);
e.Graphics.DrawPolygon(new Pen(Color.Black, 1), ptPoly);
// define a cor azul para preenchimento da circunferência
corFundo = new System.Drawing.SolidBrush(Color.Blue);
// desenha a circunferência
e.Graphics.FillEllipse(corFundo, (float)(pnl.Width / 2 - r),
    (float)(pnl.Height / 2 - r), (float)(2 * r), (float)(2 * r));
e.Graphics.DrawEllipse(new Pen(Color.Black, 1),
    (float)(pnl.Width / 2 - r), (float)(pnl.Height / 2 - r),
    (float)(2 * r), (float)(2 * r));
// define o preenchimento branco da faixa horizontal
// dentro da circunferência
corFundo = new System.Drawing.SolidBrush(Color.White);
// desenha o retângulo da faixa
e.Graphics.FillRectangle(corFundo, (float)(pnl.Width / 2 - r),
    (float)(pnl.Height / 2 - 7), (float)(2 * r), 14);
// calcula a largura do texto para poder centralizar
w = e.Graphics.MeasureString("ORDEM E PROGRESSO",
    drawFont).Width;
// define a cor da letra do texto
corFundo = new System.Drawing.SolidBrush(Color.Green);
// escreve o texto centralizado na faixa
e.Graphics.DrawString("ORDEM E PROGRESSO", drawFont, corFundo,
    (float)((pnl.Width - w) / 2), (float)(pnl.Height / 2 - 5));
drawFont.Dispose();
}
```

8.2. Delegates

Um delegate serve para declarar uma variável que vai armazenar um método. Para entender como um delegate funciona, abra o projeto que está na pasta **Cap_08\2_Lambda_Ex2**:



- **A** - Vai gerar uma sequência numérica de 1 até 100 e dividir ou multiplicar cada elemento por 2, de acordo com o **RadioButton** selecionado. O segundo botão vai fazer isso usando a expressão lambda. O resultado deverá ser mostrado no ListBox à direita;
- **B** - Vai gerar uma sequência de 0 a 100 e calcular o quadrado de cada elemento. O resultado deverá ser mostrado no ListBox à direita;
- **C** - Vai gerar uma sequência de 0 a 360 e calcular o seno do ângulo correspondente (**Math.Sin(anguloEmRadianos)**). O resultado deverá ser mostrado no ListBox à direita;
- **D** - Vai gerar uma sequência de 1 a 100 e calcular 1 dividido por cada elemento. O resultado deverá ser mostrado no ListBox à direita;

- E - Vai gerar uma sequência de 100 até 1 e calcular 1 dividido por cada elemento. O resultado deverá ser mostrado no ListBox à direita.

Sem o uso de delegate, cada botão teria que ter um loop para efetuar o cálculo (que é diferente para cada botão) dentro do corpo do loop. Com delegate, o procedimento fica mais simples:

```
// declara a sintaxe (ou assinatura) de um método
delegate double FuncDeX(double x);

// cria métodos que atendem ao delegate FuncDeX
double divide(double n)
{
    return n / 2;
}

double multiplica(double n)
{
    return n * 2;
}
```

O delegate declarado nesse exemplo define um método que recebe um dado do tipo double e retorna com um dado do tipo double, da mesma forma que os métodos **divide** e **multiplica** declarados na sequência. Isso também é válido para qualquer outro método que tenha essa mesma assinatura. Quando declararmos uma variável do tipo **FuncDeX**, ela poderá receber o método **divide** ou o método **multiplica**:

```
FuncDeX f = divide;
```

Também podemos fazer com que um método receba uma variável que armazena outro método:

```
// rotina que recebe a f que é do tipo FuncDeX. Esta variável aponta para
// um método que respeite a assinatura criada pelo delegate. Pode apontar
// para o método divide, multiplica ou qualquer outro que receba como
// parâmetro um dado do tipo double e retorne com double
void Calcula(FuncDeX f, double inicio, double fim, double incr)
```

C# - Módulo I

```
{  
    lbx.Items.Clear();  
    double ctdr = inicio;  
    // loop para gerar os valores de Y  
    while (ctdr <= fim)  
    {  
        // calcula Y de acordo com o método apontado pela variável f  
        double y = f( ctdr );  
        // mostra os valores de X e Y no listBox  
        lbx.Items.Add ( "f( " + ctdr + " ) = " + y.ToString("0.000"));  
        ctdr += incr;  
    }  
}
```

1. Crie o evento **Click** do primeiro botão **Divide ou Multiplica**:

```
// Executa o método calcula utilizando divide ou multiplica  
// dependendo do RadioButton selecionado  
private void btnDivMult1_Click(object sender, EventArgs e)  
{  
    // declara a variável para armazenar o método  
    FuncDeX f;  
    // armazena o método na variável dependendo do  
    // RadioButton selecionado  
    if (rbDivide.Checked) f = divide;  
    else f = multiplica;  
    // executa o método Calcula passando para ele a variável f  
    Calcula(f, 1, 100, 1);  
}
```

2. Crie o evento **Click** do segundo botão **Divide ou Multiplica**. Como os métodos **divide** e **multiplica** retornam diretamente com o resultado de um cálculo, podemos substituí-los por uma expressão lambda:

```
(listaDeParametros) => expressãoDeRetorno
```

```

private void btnDivMult2_Click(object sender, EventArgs e)
{
    FuncDeX f;

    if (rbDivide.Checked) f = (x) => x / 2;
    else f = (x) => x * 2;

    Calcula(f, 1, 100, 1);
}

```

3. Crie o evento **Click** do botão para calcular o quadrado do número. Neste caso, podemos passar a expressão diretamente para o método **Calcula**:

```

// passa a expressão lambda diretamente como parâmetro
// para o método calcula
private void btnQuadrado_Click(object sender, EventArgs e)
{
    Calcula( (n) => n * n, 1, 100, 1);
}

```

4. Crie o evento **Click** do botão para calcular o seno do ângulo;

```

private void btnSeno_Click(object sender, EventArgs e)
{
    Calcula((a) => Math.Sin(a * 2 * Math.PI / 360), 0, 360 , 1);
}

```

5. Crie o evento **Click** do botão para calcular $1 / x$ de 1 até 100;

```

private void btnDivX1_Click(object sender, EventArgs e)
{
    Calcula2( (x) => 1 / x, 1, 100, 1);
}

```

6. Crie o evento **Click** do botão para calcular $1 / x$ de 1 até 100;

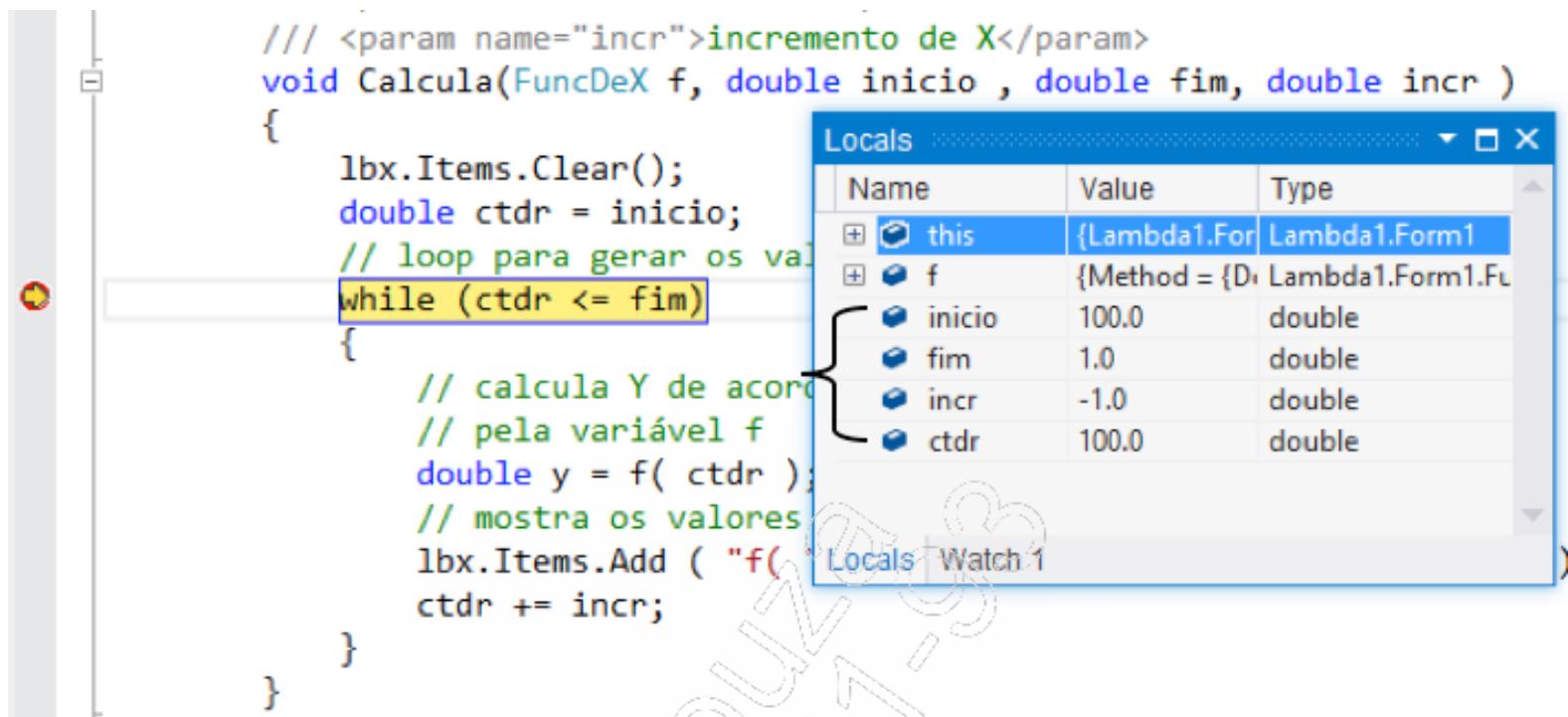
```

private void btnDivX2_Click(object sender, EventArgs e)
{
    Calcula2( (x) => 1 / x, 100, 1, -1);
}

```

C# - Módulo I

7. Teste este botão. Não funcionou! Se você depurar o programa, vai ver que dentro do método **calcula**, ele não entra no loop;



8. Note que o valor do contador (100) já é maior que a variável fim (1), então ele nem entra no Loop. Para fazer a contagem regressiva, a condição deveria ser:

```
while (ctdr >= fim)
```

9. No entanto, a condição do while também é uma expressão, no caso, do tipo Boolean. Portanto, ela pode ser armazenada em uma variável, desde que criemos um delegate com a sua assinatura:

```
// declara delegate para condição do loop while
// métodos declarados do tipo CondWhile irão receber
// 2 parâmetros double, compará-los e retornar true ou false
delegate bool CondWhile(double cont, double limite);
```

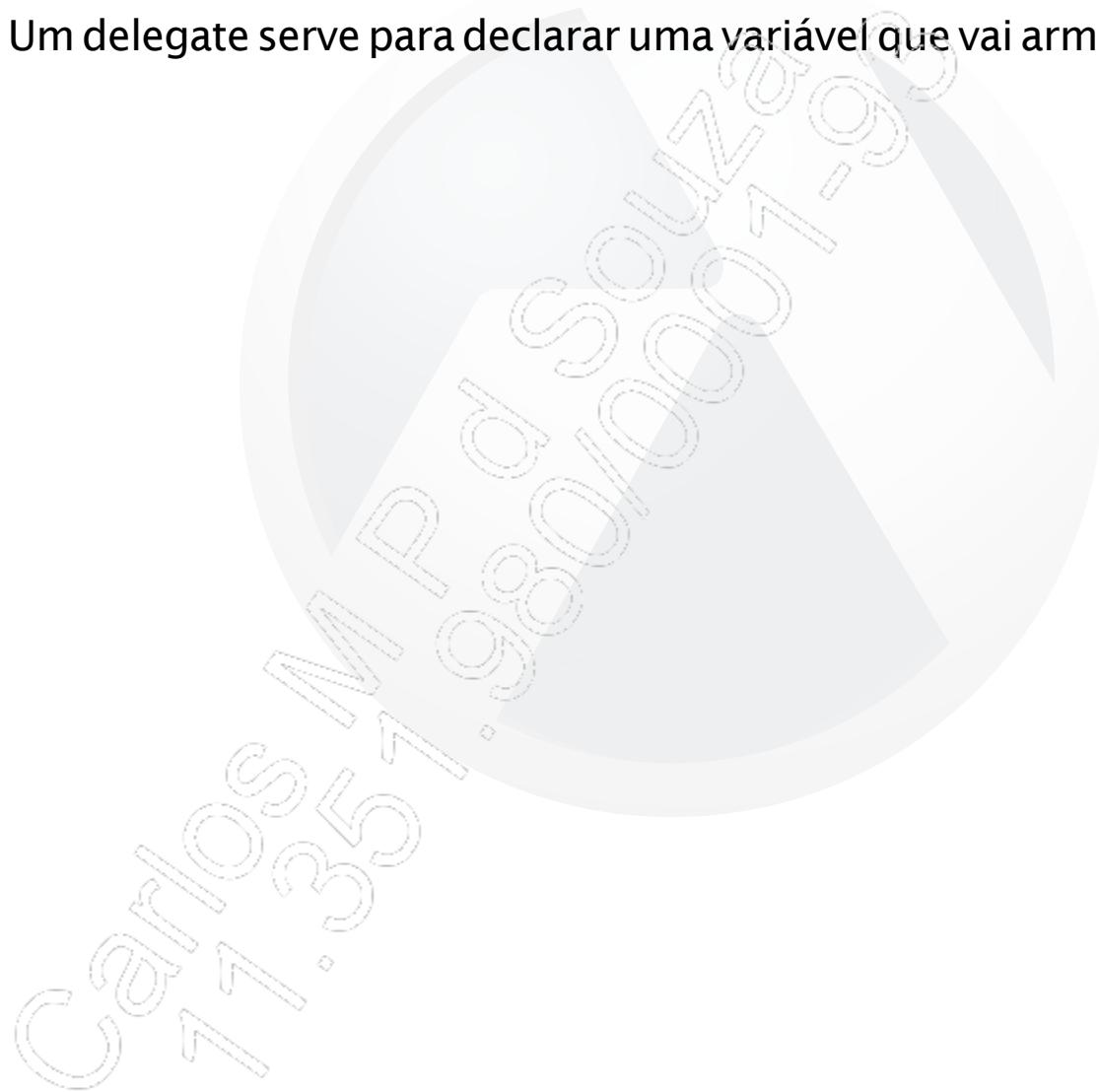
10. Utilize o novo método **Calcula**, que pode fazer a contagem tanto crescente quanto decrescente:

```
void Calcula2(FuncDeX f, double inicio, double fim, double incr)
{
    lbx.Items.Clear();
    // declara variável para armazenar a expressão
    // que resulta na condição do While
    CondWhile cond;
    // se o inicio for menor que o fim, a contagem deverá ser progressiva
    // e o loop deve prosseguir enquanto ctdr <= fim
    if (inicio < fim) cond = (cont, limite) => cont <= limite;
    // caso contrário, a contagem será regressiva
    // e o loop deve prosseguir enquanto ctdr >= fim
    else cond = (cont, limite) => cont >= limite;
    double ctdr = inicio;
    // executa o loop while utilizando como condição
    // a expressão atribuída à variável cond. Passamos
    // para a expressão o valor de ctdr (contador em cont)
    // e fim (limite de contagem em limite)
    while (cond(ctdr, fim))
    {
        double y = f(ctdr);
        lbx.Items.Add("f( " + ctdr.ToString("0.000") + " ) = " +
                      y.ToString("0.000"));
        ctdr += incr;
    }
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Enumeração é um tipo de dado usado para declarar variáveis, propriedades e métodos que contêm um valor numérico;
- Se uma variável do tipo **DayOfWeek** for declarada, ela aceitará somente uma das opções contidas no tipo enumerado;
- Um delegate serve para declarar uma variável que vai armazenar um método.



8

Enumerações e Delegates

Teste seus conhecimentos

Carlos M. Góes
77.351-9007-03
SOUZA



IMPACTA
EDITORA

1. Qual declaração de método pode completar adequadamente a lacuna a seguir?

```
delegate string CalcData( DateTime dt );  
CalcData cd = _____
```

- a) string DiaSemana(void dt)
- b) DateTime DiaSemana(string dt)
- c) void DiaSemana(DateTime dt)
- d) string DiaSemana(DateTime dt)
- e) void string DiaSemana(DateTime dt)

2. O que podemos afirmar sobre delegate?

- a) Serve para armazenar um método.
- b) Cria um tipo de dado com o qual podemos declarar uma variável que armazena um método ou expressão lambda.
- c) Armazena um método ou expressão lambda.
- d) Cria um tipo de dado com o qual podemos declarar uma variável que armazena um método void (não retorna valor).
- e) Nenhuma das alternativas anteriores está correta.

3. Qual valor será mostrado no label1?

```
enum numeros { Primeiro, Segundo, Terceiro, Quarto, Quinto };

private void button1_Click(object sender, EventArgs e)
{
    numeros n = numeros.Terceiro;
    label1.Text = ((int)n).ToString();

}
```

- a) 0
- b) 1
- c) 2
- d) 3
- e) 4

4. Qual valor será mostrado no label1?

```
enum numeros { Primeiro = 1, Segundo, Terceiro, Quarto, Quinto };

private void button1_Click(object sender, EventArgs e)
{
    numeros n = numeros.Terceiro;
    label1.Text = ((int)n).ToString();

}
```

- a) 0
- b) 1
- c) 2
- d) 3
- e) 4

5. Qual valor será mostrado no label1?

```
enum numeros { Primeiro , Segundo = 3, Terceiro, Quarto, Quinto };

private void button1_Click(object sender, EventArgs e)
{
    numeros n = numeros.Terceiro;
    label1.Text = ((int)n).ToString();
}
```

- a) 0
- b) 1
- c) 2
- d) 3
- e) 4

8

Enumerações e Delegates

Mãos à obra!

Carlos M. Gómez
77.357.990-007-003
SOUZA

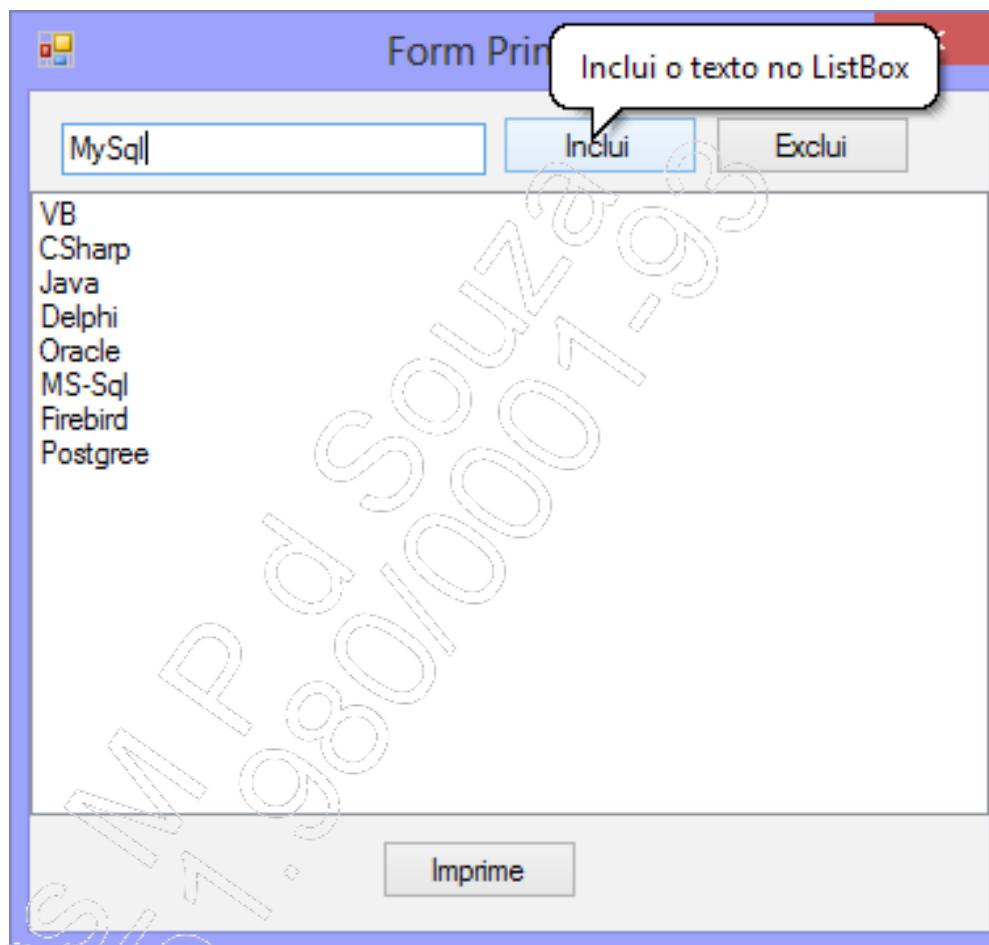


IMPACTA
EDITORA

Laboratório 1

A - Criando um formulário que imprime o conteúdo de qualquer ListBox

1. Abra o projeto **PrintListBox** da pasta **Cap_08\3_PrintListBox_Lab1**. Neste projeto, você criará um formulário capaz de imprimir o conteúdo de qualquer ListBox. Na tela principal do projeto (**FormPrincipal**), existe o ListBox que será usado para teste;



2. Observe os métodos existentes até agora. Não há novidade por enquanto;

```
private void incluiButton_Click(object sender, EventArgs e)
{
    lbx.Items.Add(itemTextBox.Text);
}

private void btnExclui_Click(object sender, EventArgs e)
{
    if (lbx.SelectedIndex >= 0)
        lbx.Items.RemoveAt(lbx.SelectedIndex);
}
```

```
private void imprimeButton_Click(object sender, EventArgs e)
{
    // cria o form que imprime ListBox
    FormPrintListBox frm = new FormPrintListBox(lbx);
    // mostra o formulário
    frm.ShowDialog();
}
```

3. Um segundo formulário (**FormPrintListBox**) fará a impressão de qualquer **ListBox**, porque ele recebe o **ListBox** como parâmetro no seu método construtor;

```
namespace PrintListBox
{
    public partial class FormPrintListBox : Form
    {

        // variável para armazenar o ListBox que será impresso
        ListBox lbx;

        // construtor recebe o ListBox que será impresso
        public FormPrintListBox(ListBox lbx)
        {
            InitializeComponent();
            // transfere o dado recebido como parâmetro para a variável da classe
            this.lbx = lbx;
        }
}
```

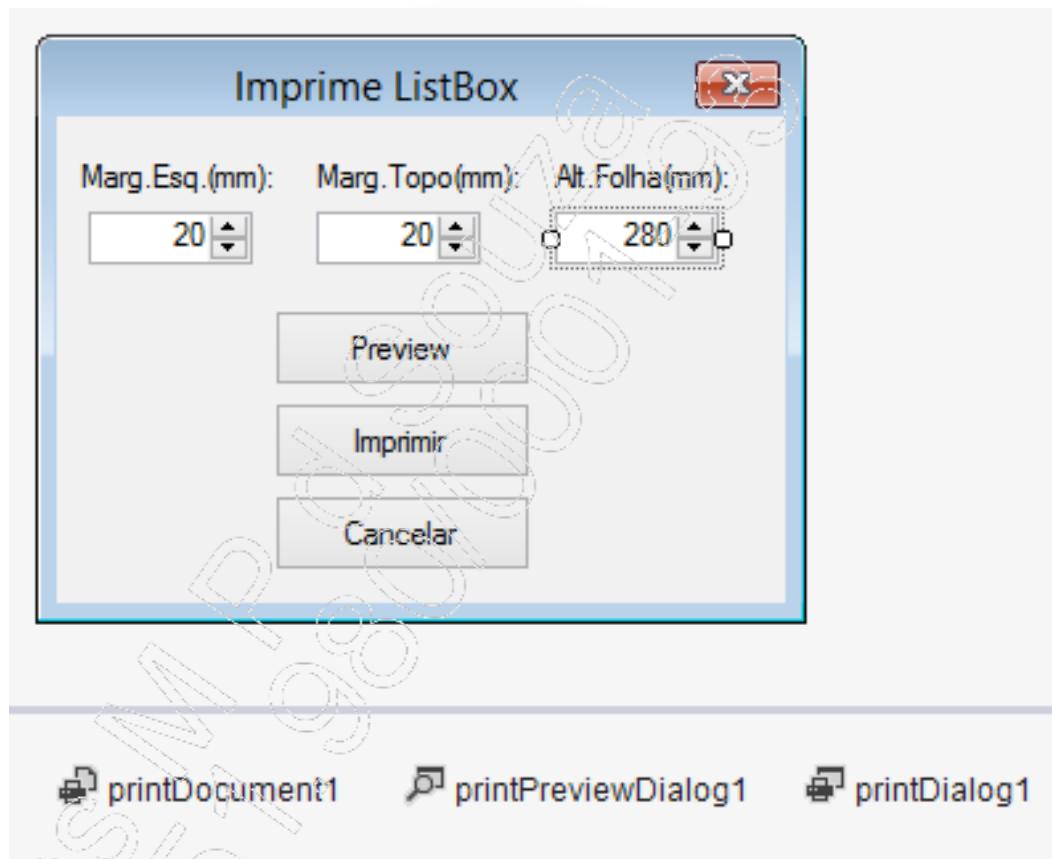
4. Nesse formulário, foram colocados os componentes necessários para impressão:

- **PrintDocument**;
- **PrintPreviewDialog**;
- **PrintDialog**.

5. Você também deve configurar três componentes **NumericUpDown**:

- Margem esquerda (em milímetros);
- Margem do topo (em milímetros);
- Altura da folha (em milímetros).

6. Os botões da janela a seguir farão as tarefas de impressão e preview de impressão. Para isso, confira os códigos adiante:



- O evento **PrintPage** também já está pronto, e não há novidades nele:

```
// contadores de linha e de página
int ctLin, ctPag;

// evento PrintPage responsável pela impressão de cada página do
// documento
private void printDocument1_PrintPage(object sender, PrintPageEventArgs
e)
{
    // variável para posição da linha ( Y )
    int yPos = (int)margTopoUpDown.Value;
    // cor da letra
```

```

Brush br = Brushes.Black;
// unidade de medida para X e Y
e.Graphics.GraphicsUnit = GraphicsUnit.Millimeter;

// medir o tamanho da letra de acordo com a fonte do texto
SizeF tLetra = e.Graphics.MeasureString("X", lbx.Font);
// enquanto não acabar a folha e não acabarem os itens
while (ctLin <= margTopoUpDown.Value && ctLin < lbx.Items.Count)
{
    // imprimir o item do ListBox e incrementar o contador de linha
    e.Graphics.DrawString(lbx.Items[ctLin++].ToString(), lbx.Font,
        br, (float)margEsqUpDown.Value, yPos);
    // incrementar posição Y
    yPos += (int)tLetra.Height + 2;
}

// incrementa o contador de páginas
ctPag++;
// se true, salta para próxima página e volta para
// este método para imprimir nova página
// se false, finaliza a impressão
e.HasMorePages = ctLin < lbx.Items.Count;
}

```

- Configure os botões para preview de impressão e impressão:

```

// gera o preview de impressão
private void previewButton_Click(object sender, EventArgs e)
{
    ctLin = 0;
    ctPag = 1;

    printPreviewDialog1.WindowState = FormWindowState.Maximized;
    printPreviewDialog1.ShowDialog();
}

// imprime na impressora selecionada em printDialog1
private void imprimirButton_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
        ctLin = 0;
        ctPag = 1;
        printDocument1.Print();
    }
}

```

Onde é que entram os delegates e eventos?

Lembre-se de que este formulário deve ter a função de imprimir o conteúdo de qualquer ListBox, configurando as margens, tamanho da folha e usando a mesma fonte usada no ListBox original. Existem ainda algumas outras variáveis:

- Imagine que a pessoa que imprime o seu ListBox usando este Form queira imprimir um título ou um cabeçalho antes dos dados;
- Imagine que este cabeçalho possa ter uma ou mais linhas, com imagem, numeração de página, etc;
- Imagine também que a pessoa queira imprimir um rodapé de página contendo o texto que ele bem entender, com numeração de páginas ou não.

Percebe que é impossível para o **FormPrintListBox** parametrizar tudo isso?

A melhor situação neste caso é criar dois eventos:

- **ImprimeCabecalho**;
- **ImprimeRodape**.

A classe que usa o **FormPrintListBox** – em nosso exemplo, **FormPrincipal** – é que vai programar estes eventos como ela bem entender.

FormPrintDialog

```
// evento PrintPage responsável pela impressão de cada página do documento
private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
    // variável para posição da linha ( Y )
    int yPos = (int)margTopUpDown.Value;
    // cor da letra
    Brush br = Brushes.Black;
    // unidade de medida para X e Y
    e.Graphics.GraphicsUnit = GraphicsUnit.Millimeter;
    // medir o tamanho da letra de acordo com a fonte do texto
    SizeF tletra = e.Graphics.MeasureString("X", lbx.Font);
    // enquanto não acabar a folha e não acabarem os itens
    while (ctlIn <= margTopUpDown.Value && ctlIn < lbx.Items.Count)
    {
        // imprimir o item do ListBox e incrementar o contador de linha
        e.Graphics.DrawString(lbx.Items[ctlIn++].ToString(), lbx.Font,
            br, (float)margEsquedoUp.Value, yPos);
        // incrementar posição Y
        yPos += (int)tletra.height + 2;
    }
    // incrementa o contador de páginas
    ctPag++;
    // se true, salta para próxima página e volta para
    // este método para imprimir nova página
    // se false, finaliza a impressão
    e.HasMorePages = ctlIn < lbx.Items.Count;
}
```

The diagram shows two code snippets side-by-side. Arrows point from specific lines of code in the left snippet to corresponding event handler definitions in the right snippet. One arrow points from the line 'e.Graphics.DrawString(lbx.Items[ctlIn++].ToString(), lbx.Font, br, (float)margEsquedoUp.Value, yPos);' to the 'frm_ImprimeCabecalho' method. Another arrow points from the line 'yPos += (int)tletra.height + 2;' to the 'frm_ImprimeRodape' method.

```
// cabecalho que aparecerá antes do primeiro item do ListBox
int frm_ImprimeCabecalho(System.Drawing.Printing.PrintPageEventArgs e, int ctPag, int yPos)
{
}

// rodapé que aparecerá depois do último item do ListBox
// neste caso mostraremos o número da página
int frm_ImprimeRodape(System.Drawing.Printing.PrintPageEventArgs e, int ctPag, int yPos)
{
}
```

FormPrincipal

Esta é uma utilização típica de delegates e eventos.

7. Selecione **FormPrintDialog** e crie um delegate que define a assinatura dos métodos para impressão do cabeçalho e do rodapé:

```
public partial class FormPrintListBox : Form
{
    /// <summary>
    /// cria delegate com a assinatura dos métodos para impressão do
    /// cabeçalho e do rodapé
    /// </summary>
    /// <param name="e">Possui os recursos para impressão</param>
    /// <param name="ctPag">Contador de páginas</param>
    /// <param name="yPos">Posição Y para atual</param>
    /// <returns>
    /// Nova posição Y após a impressão do cabeçalho ou rodapé</returns>
    public delegate int CabecalhoRodape(PrintPageEventArgs e,
                                         int ctPag, int yPos);
```

8. Agora crie os dois eventos;

```
/// <summary>
/// Evento para impressão do cabeçalho de página
/// </summary>
public event CabecalhoRodape ImprimeCabecalho;
/// <summary>
/// Evento para impressão do rodapé de página
/// </summary>
public event CabecalhoRodape ImprimeRodape;
```

C# - Módulo I

9. No evento **PrintPage**, verifique se os eventos anteriormente criados apontam para algum método e então execute-os. Antes de iniciar o loop de impressão de linhas, teste se **ImprimeCabecalho** contém algum método:

```
// evento PrintPage responsável pela impressão de cada página do
private void printDocument1_PrintPage(object sender, PrintPageEventArgs
e)
{
    // variável para posição da linha ( Y )
    int yPos = (int)margTopoUpDown.Value;
    // cor da letra
    Brush br = Brushes.Black;
    // unidade de medida para X e Y
    e.Graphics.GraphicsUnit = GraphicsUnit.Millimeter;

    // se existir método para o evento ImprimeCabecalho
    if (ImprimeCabecalho != null)
    {
        // executar o método e recuperar o novo valor de yPos
        yPos = ImprimeCabecalho(e, ctPag, yPos);
    }
}
```

10. Ao final do loop, faça o mesmo em relação ao rodapé de página:

```
while (ctLin <= margTopoUpDown.Value && ctLin < lbx.Items.Count)
{
    // imprimir o item do ListBox e incrementar o contador de linha
    e.Graphics.DrawString(lbx.Items[ctLin++].ToString(), lbx.Font,
        br, (float)margEsqUpDown.Value, yPos);
    // incrementar posição Y
    yPos += (int)tLetra.Height + 2;

    // se existir método para o evento ImprimeRodape
    if (ImprimeRodape != null)
    {
        yPos = ImprimeRodape(e, ctPag, yPos);
    }

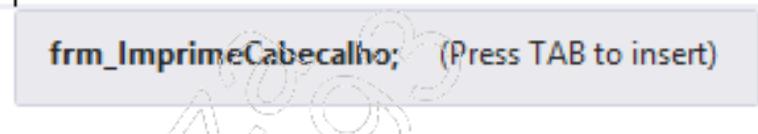
    // incrementa o contador de páginas
    ctPag++;
}
```

11. Agora você pode retornar ao **FormPrincipal** e testar estes eventos:

- Evento **Click** do botão **imprimeButton**. Digite até o ponto mostrado na imagem a seguir e depois pressione a tecla TAB duas vezes;

```
private void imprimeButton_Click(object sender, EventArgs e)
{
    // cria o form que imprime ListBox
    FormPrintListBox frm = new FormPrintListBox(lbx);
    // configura os eventos para impressão do cabeçalho e do rodapé
    frm.ImprimeCabecalho +=

    // mostra o formulário
    frm.ShowDialog();
}
```



O resultado será:

```
private void imprimeButton_Click(object sender, EventArgs e)
{
    // cria o form que imprime ListBox
    FormPrintListBox frm = new FormPrintListBox(lbx);
    // configura os eventos para impressão do cabeçalho e do rodapé
    frm.ImprimeCabecalho +=
        new FormPrintListBox.CabecalhoRodape(frm_ImprimeCabecalho);
    // mostra o formulário
    frm.ShowDialog();
}

int frm_ImprimeCabecalho(System.Drawing.Printing.PrintPageEventArgs e,
                        int ctPag, int yPos)
{
    throw new NotImplementedException();
}
```

C# - Módulo I

- Pressione TAB duas vezes quando estiver no evento **ImprimeRodape** e depois altere os métodos gerados como mostra o código a seguir:

```
private void imprimeButton_Click(object sender, EventArgs e)
{
    // cria o form que imprime ListBox
    FormPrintListBox frm = new FormPrintListBox(lbx);
    // configura os eventos para impressão do cabeçalho e do rodapé
    frm.ImprimeCabecalho +=
        new FormPrintListBox.CabecalhoRodape(frm_ImprimeCabecalho);
    frm.ImprimeRodape +=
        new FormPrintListBox.CabecalhoRodape(frm_ImprimeRodape);
    // mostra o formulário
    frm.ShowDialog();
}

// cabeçalho que aparecerá antes do primeiro item do ListBox
int frm_ImprimeCabecalho(System.Drawing.Printing.PrintEventArgs e,
int ctPag, int yPos)
{
    // define título do cabeçalho, neste caso somente 1 título
    string titulo = "LINGUAGENS E BANCOS DE DADOS";
    // define o tipo de letra que será usado na impressão
    Font f = new Font("Arial", 12, FontStyle.Bold);
    // calcula o tamanho (largura e altura) do título de
    // acordo com a fonte atual
    SizeF t = e.Graphics.MeasureString(titulo, f);
    // imprime o título centralizado
    e.Graphics.DrawString(titulo, f, Brushes.Black,
        (float)(200 - t.Width) / 2, yPos);
    // retorna com o novo yPos, somando no inicial a
    // altura da letra mais 3 milímetros
    return (int)(yPos + t.Height + 3);
}

// rodapé que aparecerá depois do último item do ListBox
// neste caso mostraremos o número da página
int frm_ImprimeRodape(System.Drawing.Printing.PrintEventArgs e, int
ctPag, int yPos)
{
    // fonte que será usada
    Font f = new Font("Arial", 12, FontStyle.Bold);
    // imprime o número da página
    e.Graphics.DrawString("Pág.: " + ctPag, f, Brushes.Black,
        190, yPos + 5);
    // como o rodapé é a última coisa que será impressa na página,
    // o yPos retornado não será usado
    return 0;
}
```

Classes e estruturas

9

- ✓ Semelhanças entre classes e estruturas;
- ✓ Diferenças entre classes e estruturas;
- ✓ Classes;
- ✓ Classes static;
- ✓ Métodos de extensão;
- ✓ Herança.



IMPACTA
EDITORA

9.1. Introdução

Uma classe, assim como uma estrutura, define as características dos objetos pertencentes a ela. Essas características ou atributos são suas propriedades (dados que ela manipula) e seus métodos (ações que ela executa).

A utilidade de ambas, classe e estrutura, é reunir em um único elemento todas as informações necessárias para resolver um dado problema. Imagine que precisamos calcular todos os impostos envolvidos na emissão de uma nota fiscal (PIS, COFINS, IPI, ICMS etc...). Podemos criar uma classe ou uma estrutura para este fim. Ela reuniria os dados (propriedades) e ações (métodos) para resolver o problema e nos dar as respostas. Isto também é chamado de “encapsulamento”. As diferenças entre classes e estruturas serão mostradas mais adiante.

Os itens a seguir são válidos tanto para classes quanto para estruturas. Vamos conferir a descrição de cada um deles:

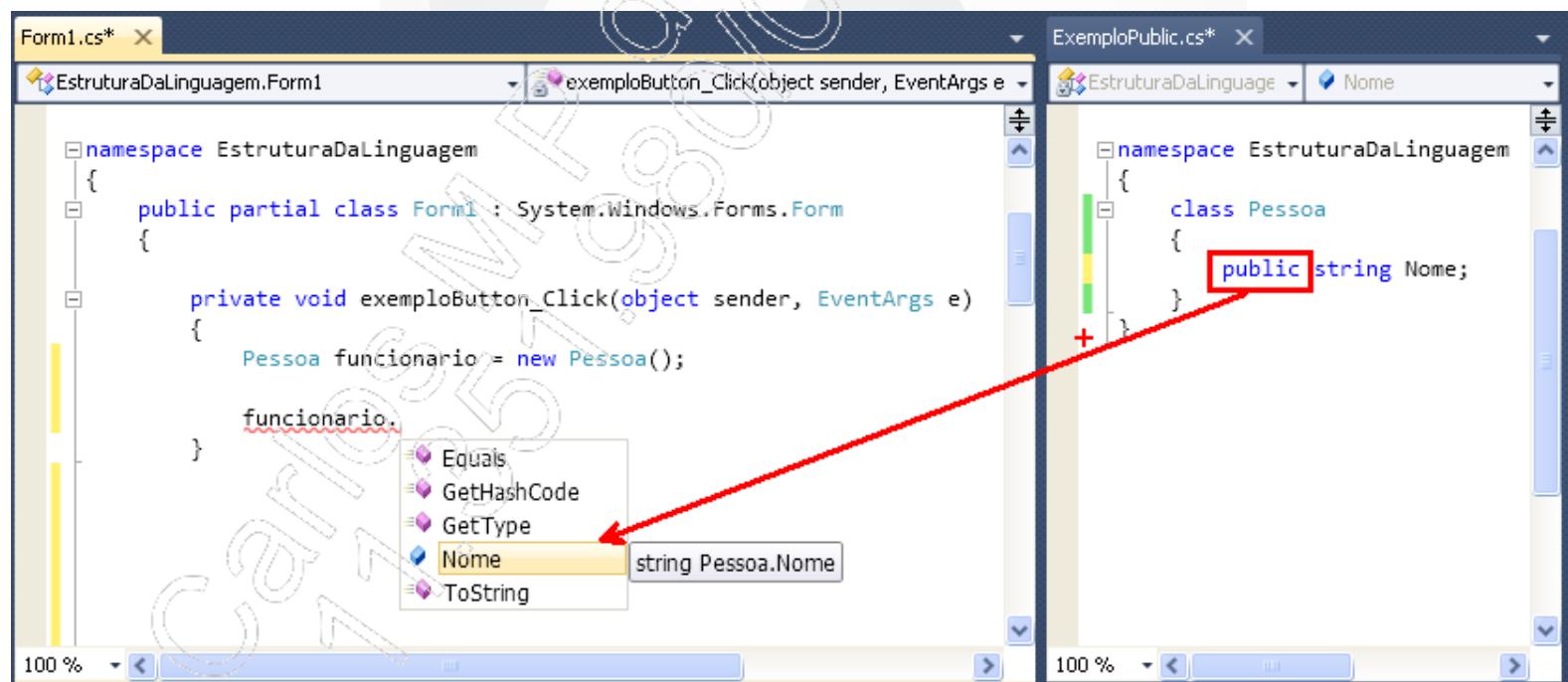
- **Campos:** São variáveis declaradas dentro das classes que armazenam os conteúdos de cada propriedade. As variáveis “campo” são visíveis apenas dentro da classe (private);
- **Propriedades:** São como variáveis que armazenam os dados necessários para que a classe possa cumprir sua função. Normalmente, o dado contido na propriedade fica armazenado em uma variável “campo” e é lido ou gravado nesta variável através dos métodos **Get** e **Set**;
- **Métodos:** São rotinas que realizam tarefas. Eles podem ou não retornar valores e podem ou não receber parâmetros;
- **Construtores:** Métodos responsáveis pela criação de um objeto de uma determinada classe. Uma classe pode ter vários construtores e todos eles serão públicos e terão o mesmo nome da classe;
- **Eventos:** É uma forma de permitir que uma classe execute um método que está escrito em outra classe. Quando escrevemos um método para o evento **Click** de um botão, este método é escrito no formulário, mas quem o executa é o Button.

Um evento é muito semelhante a uma propriedade, mas o que ele armazena é um método.

Os modificadores de acesso definem a visibilidade de campo, propriedade, evento e método de uma classe. A seguir, temos uma tabela em que estão descritos os tipos de modificadores de acesso que a linguagem C# oferece:

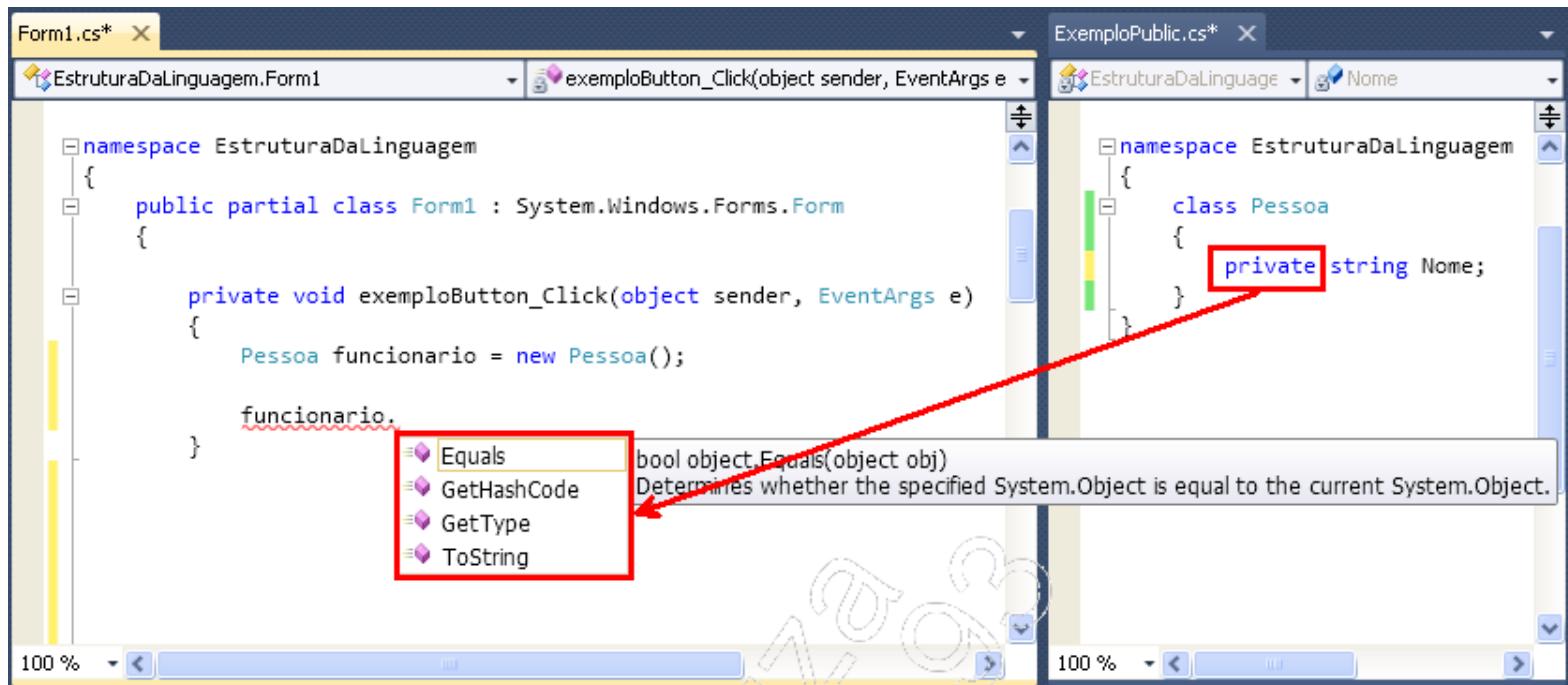
Modificador	Descrição
Public	O acesso é livre quando utilizamos esse modificador.
Protected	Ao utilizarmos esse modificador, apenas a classe que contém o modificador e os tipos derivados dessa classe possuem acesso.
Internal	Esse modificador limita o acesso apenas ao assembly atual.
Protected internal	Quando utilizamos esse modificador, o acesso é limitado ao assembly atual e aos tipos derivados da classe que contém o modificador.
Private	Quando utilizamos esse modificador, apenas o tipo que contém o modificador tem o acesso. Por padrão, private é a visibilidade definida para métodos e atributos em uma classe.

- **Modificador public**



C# - Módulo I

- Modificador private



- Modificador protected

The screenshot shows three code files. Pessoas.cs defines a class Pessoa with a private string field _nome and a public string property Nome. It also contains a protected string method Meditar() which returns "Meditando". Pedido.cs defines a class Pedido with a Pessoa cliente field and a public void method EmitirPedido(). This method calls the cliente.Nome property. Funcionario.cs defines a class Funcionario that inherits from Pessoa. It has a private string field _cargo and a public string property Cargo. It also has a public string method MostrarDados() which returns a string formatted with Nome, Cargo, and the result of Meditar(). Red boxes highlight the protected method Meditar() in Pessoas.cs and the Nome property in Pedido.cs. A red arrow points from the Meditar() method in Pessoas.cs to its call in the MostrarDados() method of Funcionario.cs. Another red arrow points from the Nome property in Pedido.cs to its call in MostrarDados() of Funcionario.cs.

```
class Pessoa
{
    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    protected string Meditar()
    {
        return "Meditando";
    }
}

class Pedido
{
    Pessoa cliente = new Pessoa();

    public void EmitirPedido()
    {
        cliente.
    }
}

class Funcionario : Pessoa
{
    private string _cargo;
    public string Cargo
    {
        get { return _cargo; }
        set { _cargo = value; }
    }

    public string MostrarDados()
    {
        return string.Format("{0}\n{1}\nStatus: {2}",
            Nome, Cargo, Meditar());
    }
}
```

- Modificador internal

```

class Pessoa
{
    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    internal string Meditar()
    {
        return "Meditando";
    }
}

class Funcionario : Pessoa
{
    private string _cargo;
    public string Cargo
    {
        get { return _cargo; }
        set { _cargo = value; }
    }

    public string MostrarDados()
    {
        return string.Format("{0}\n{1}\nStatus: {2}",
            Nome, Cargo, Meditar());
    }
}

```

Pedido.cs

```

class Pedido
{
    Pessoa cliente = new Pessoa();

    public void EmitirPedido()
    {
        cliente.
    }
}

```

- Modificador **protected internal**

```
class Pessoa
{
    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    protected internal string Meditar()
    {
        return "Meditando";
    }
}

class Funcionario : Pessoa
{
    private string _cargo;
    public string Cargo
    {
        get { return _cargo; }
        set { _cargo = value; }
    }

    public string MostrarDados()
    {
        return string.Format("{0}\n{1}\nStatus: {2}",
            Nome, Cargo, Meditar());
    }
}
```

```
class Pedido
{
    Pessoa cliente = new Pessoa();

    public void EmitirPedido()
    {
        cliente.
    }
}
```

The diagram illustrates the inheritance relationship between `Pessoa` and `Funcionario`. A red box highlights the `Meditar()` method in `Pessoa`, which is then called from the `MostrarDados()` method in `Funcionario`. A red arrow points from the `Meditar()` method in `Pessoa` to the `Meditar()` method in the `Funcionario` class. Another red arrow points from the `cliente.` part of the `EmitirPedido()` code in `Pedido` to the `Meditar()` method in the `Pessoa` class, indicating that `Pedido` can also access the `Meditar()` method through its `Pessoa` reference. A red box also highlights the `Meditar()` method in the `Pessoa` class definition.

9.2. Semelhanças entre classes e estruturas

Podemos observar como são praticamente idênticas as formas de criar uma estrutura e uma classe:

- Criando uma estrutura:

```
struct StructPessoa
{
    // variável “campo” para armazenar a propriedade Nome
    private string _nome = "";
    // propriedade Nome visível externamente
    public string Nome
    {
        /* executado quando a propriedade é consultada, por ex:
         * StructPessoa p = new StructPessoa("MAGNO");
         * label1.Text = p.Nome; // consultando a propriedade
         */
        get { return _nome; }
        /* executado quando a propriedade recebe valor, por ex:
         * StructPessoa p = new StructPessoa("MAGNO");
         * p.Nome = "Carlos"; // atribuindo valor à propriedade
         */
        set { _nome = value; }
    }
    /*
     * Método construtor recebe o nome e o transfere para a
     * propriedade Nome
     * Sem ele, fariamos:
     *      StructPessoa p = new StructPessoa();
     *      p.Nome = "MAGNO";
     * Sem ele, faremos:
     *      StructPessoa p = new StructPessoa("MAGNO");
     */
    public StructPessoa(string nome)
    {
        Nome = nome;
    }
}
```

C# - Módulo I

- Criando o mesmo recurso como classe:

```
class ClassPessoa
{
    // variável “campo” para armazenar a propriedade Nome
    private string _nome;
    // propriedade Nome visível externamente
    public string Nome
    {
        /* executado quando a propriedade é consultada, por ex:
         * ClassPessoa p = new ClassPessoa("MAGNO");
         * label1.Text = p.Nome; // consultando a propriedade
         */
        get { return _nome; }
        /* executado quando a propriedade recebe valor, por ex:
         * ClassPessoa p = new ClassPessoa("MAGNO");
         * p.Nome = "Carlos"; // atribuindo valor à propriedade
         */
        set { _nome = value; }
    }
    /*
     * Método construtor recebe o nome e o transfere para a
     * propriedade Nome
     * Sem ele, fariamos:
     *     ClassPessoa p = new ClassPessoa();
     *     p.Nome = "MAGNO";
     * Sem ele, faremos:
     *     ClassPessoa p = new ClassPessoa("MAGNO");
     */
    public ClassPessoa(string nome)
    {
        Nome = nome;
    }
}
```

9.3. Diferenças entre classes e estruturas

Estruturas não possuem o recurso da herança, por consequência disso, não podem ter membros `protected`, fazer `override`, criar métodos virtuais, etc. Uma interface é um modelo para criação de uma classe. Quando criamos uma classe de acordo com o modelo determinado por uma interface, dizemos que a classe implementa a interface. Uma estrutura não pode implementar uma interface.

Uma estrutura sempre cria um construtor padrão (sem parâmetros). Já a classe somente cria o construtor padrão se não criarmos nenhum outro construtor alternativo.

A atribuição ou passagem de parâmetro de classe é por referência, já a de estruturas é por valor. Veja o exemplo:

```
private void btnTipoPorValor_Click(object sender, EventArgs e)
{
    // aloca na memória uma estrutura de nome p1
    StructPessoa p1 = new StructPessoa("MAGNO");
    // cria uma nova área na memória para armazenar p2
    // que será uma cópia de p1
    StructPessoa p2 = p1;
    lbx.Items.Clear();
    lbx.Items.Add("p1.Nome (1) = " + p1.Nome);
    lbx.Items.Add("p2.Nome (1) = " + p2.Nome);
    // alterar p2.Nome não afeta o valor contido em p1.Nome
    p2.Nome = "CARLOS";
    lbx.Items.Add("p1.Nome (2) = " + p1.Nome);
    lbx.Items.Add("p2.Nome (2) = " + p2.Nome);

}

private void btnTipoPorRef_Click(object sender, EventArgs e)
{
    // aloca na memória uma estrutura de nome p1
    ClassPessoa p1 = new ClassPessoa("MAGNO");
    // cria uma nova variável p2 que aponta para o mesmo
```

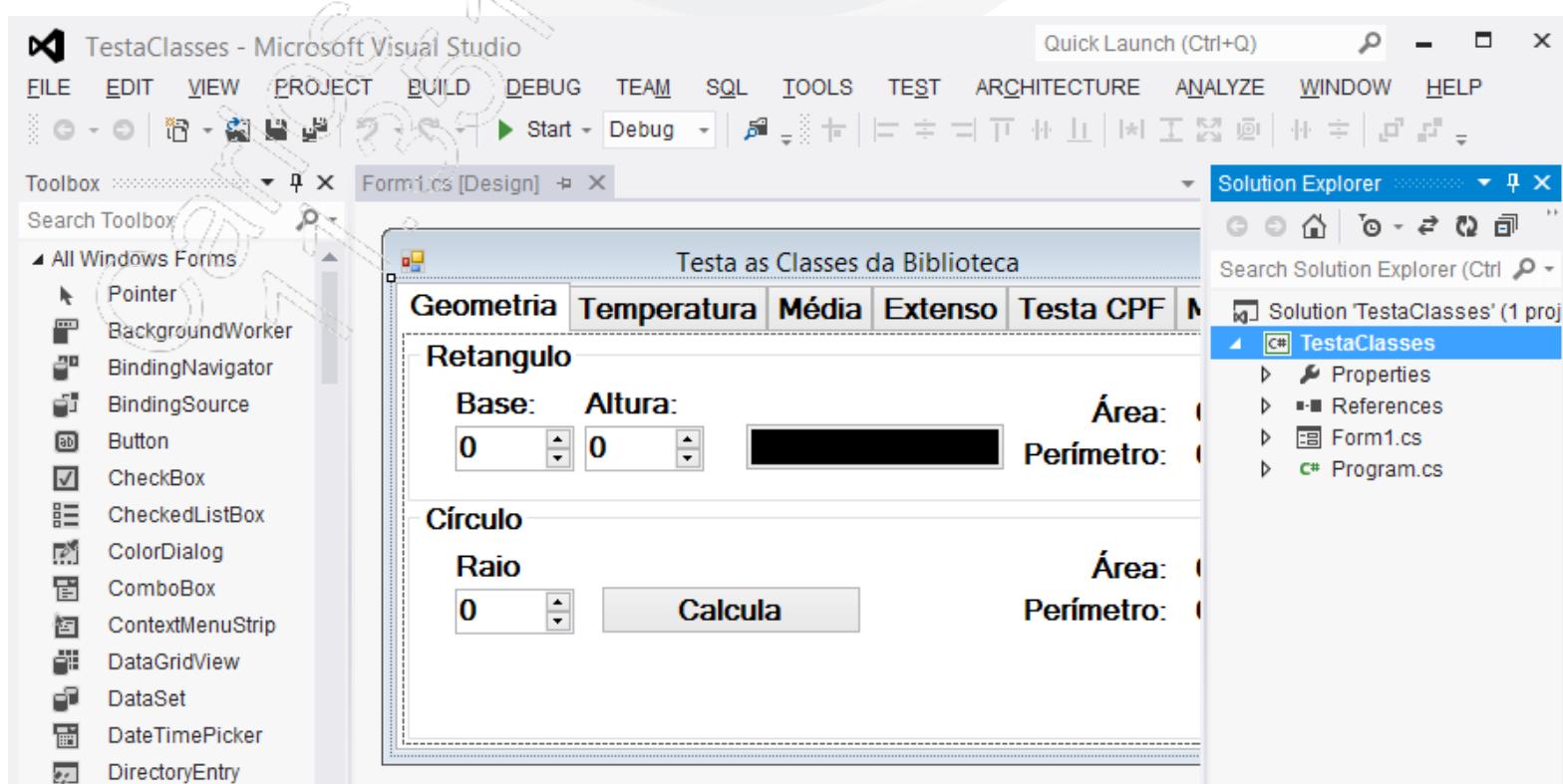
```
// endereço de memória onde está p1  
ClassPessoa p2 = p1;  
lbx.Items.Clear();  
lbx.Items.Add("p1.Nome (1) = " + p1.Nome);  
lbx.Items.Add("p2.Nome (1) = " + p2.Nome);  
// como p1 e p2 apontam para o mesmo endereço de memória  
// alterar p2.Nome afeta p1.Nome  
p2.Nome = "CARLOS";  
lbx.Items.Add("p1.Nome (2) = " + p1.Nome);  
lbx.Items.Add("p2.Nome (2) = " + p2.Nome);  
  
}
```

9.4. Classes

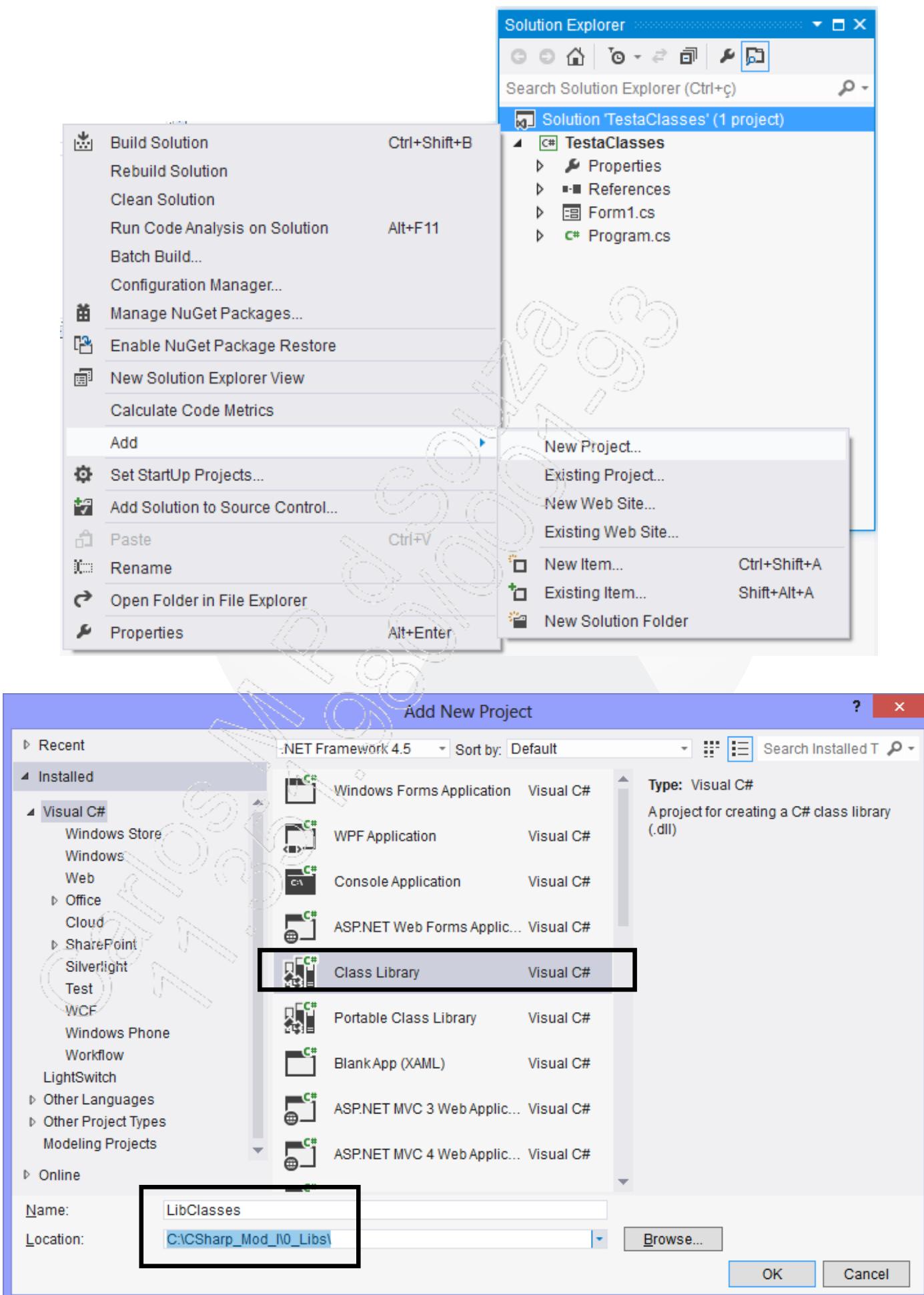
Neste tópico, teremos alguns exemplos de criação de propriedades, métodos e construtores, além do trabalho com classes.

9.4.1. Exemplo: Criação de propriedades, métodos e construtores

Neste exemplo, criaremos uma classe para fazer cálculos geométricos envolvendo um retângulo.

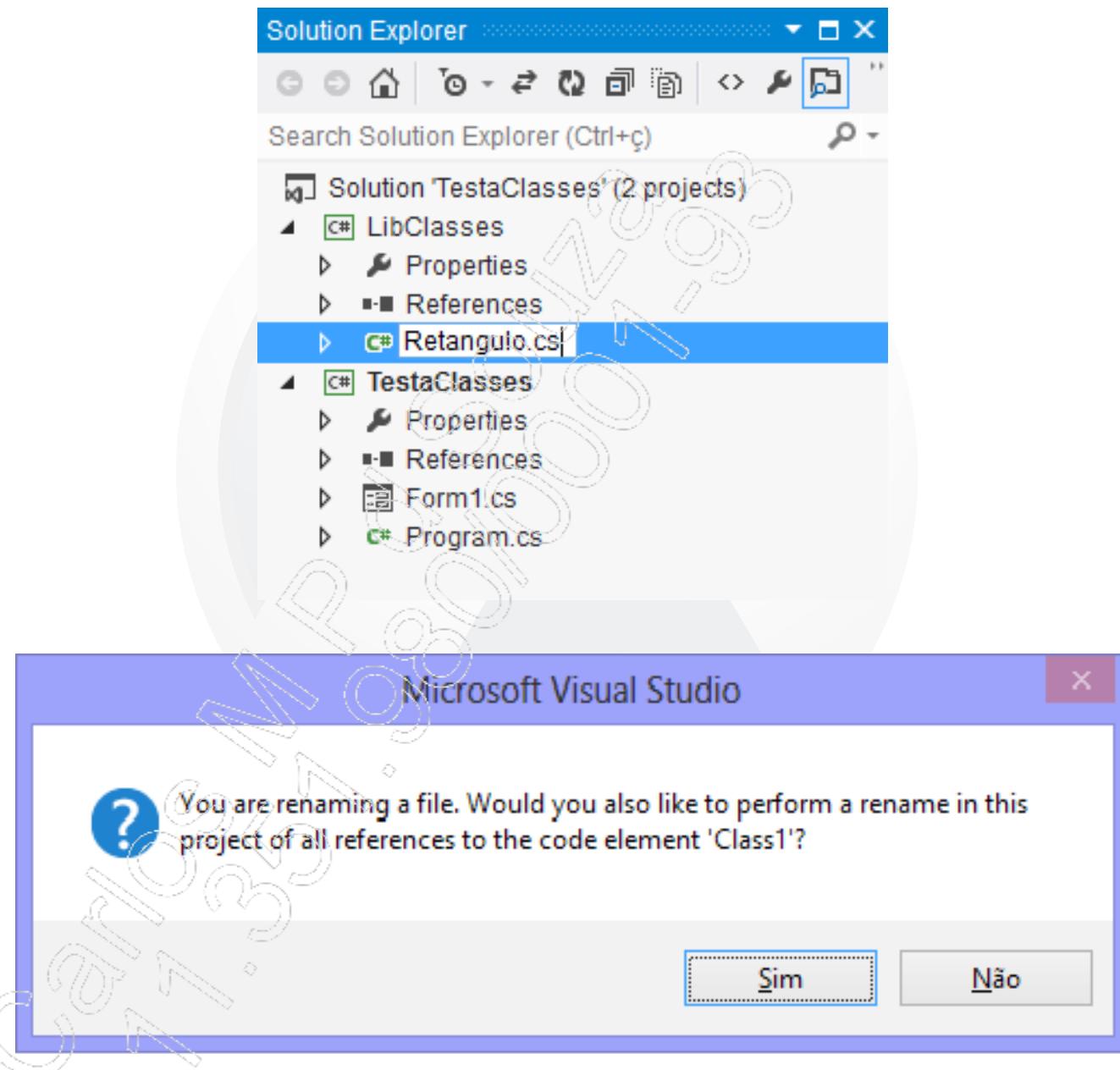


2. No **Solution Explorer**, crie um novo projeto do tipo **Class Library** chamado **LibClasses**. Como ele poderá ser usado em qualquer projeto do curso, grave-o na pasta **0_Libs_Cap09**. Veja as imagens a seguir:



3. No **Solution Explorer**, altere o nome do arquivo **Class1.cs** para **Retangulo.cs**. Em seguida, confirme a alteração do nome da classe;

! Um mesmo arquivo .cs pode conter várias classes. Neste curso, criaremos uma classe em cada arquivo, apenas para ficar mais organizado.



4. Para definir um retângulo, precisamos das medidas da base e da altura, então, vamos criar uma variável **campo** para cada uma destas propriedades:

```
public class Retangulo
{
    // variáveis private devem iniciar com letra minúscula
    // quando for uma variável para armazenar o conteúdo de uma
    // propriedade deve iniciar com underline
    private decimal _base;
    private decimal _altura;
```

5. Crie as propriedades **Base** e **Altura**:

```
// tudo que for público dentro da classe deve iniciar
// com letra maiúscula
public decimal Base
{
    /*
     * Executado quando consultar a propriedade, Por Ex:
     *
     * if (ret.Base > 10)...
     */
    get { return _base; }
    /*
     * Executado quando atribuir valor à propriedade, pr ex:
     *
     * ret.Base = 10;
     */
    set
    {
        _base = value;
    }
} // fim da prop Base

public decimal Altura
{
    get { return _altura; }
    set
    {
        _altura = value;
    }
}
```

C# - Módulo I

6. Agora vamos dar uma utilidade aos métodos SET destas duas propriedades. Suponha que, nesta classe, as propriedades **Base** e **Altura** não possam ser zero ou negativas. Neste caso, você pode fazer as seguintes alterações:

```
public decimal Base
{
    /*
     * Executado quando consultar a propriedade, Por Ex:
     *
     * if (ret.Base > 10)...
     */
    get { return _base; }
    /*
     * Executado quando atribuir valor à propriedade, pr ex:
     *
     * ret.Base = 10;
     */
    set
    {
        if (value > 0)
            _base = value;
        else
            throw new Exception("Tem que ser maior que zero");
    }
} // fim da prop Base

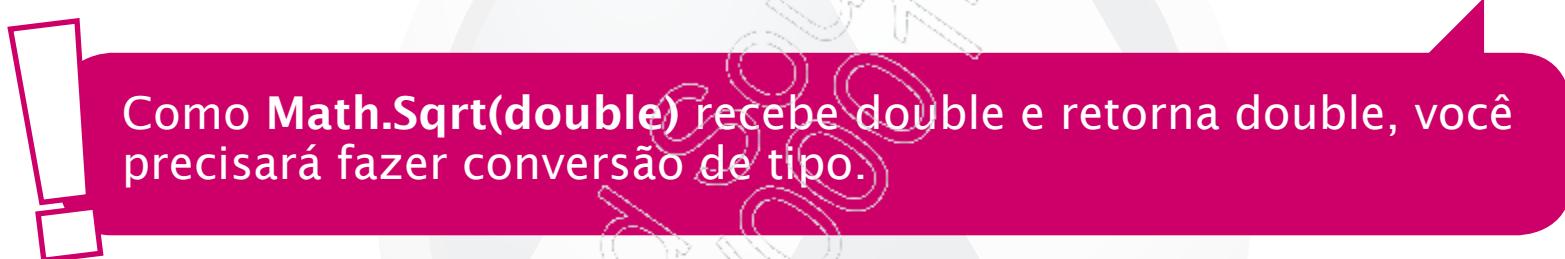
public decimal Altura
{
    get { return _altura; }
    set
    {
        if (value > 0) _altura = value;
        else throw new Exception("Tem que ser maior que zero");
    }
}
```

7. Crie o método para calcular a área do retângulo:

```
// métodos  
public decimal Area()  
{  
    return _base * _altura;  
}
```

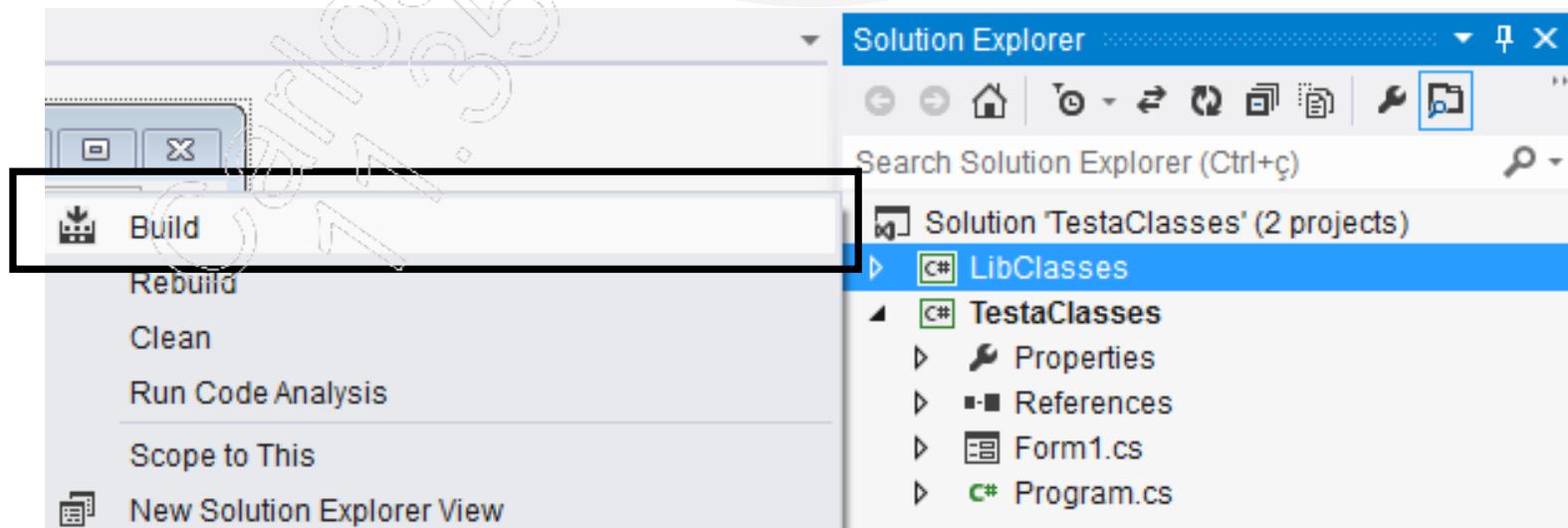
8. Sabendo que o perímetro de um retângulo corresponde a $2 * (_base + _altura)$, crie um método chamado **Perímetro()** que retorne com este valor;

9. Sabendo que a medida da diagonal de um retângulo pode ser calculada como **diagonal = Math.Sqrt(_base * _base + _altura * _altura)**, crie um método chamado **Diagonal()** que retorne com o resultado deste cálculo;



10. Para testar até este ponto:

- Compile a biblioteca **LibClasses**. Para isso, clique com o botão direito do mouse sobre **LibClasses** e escolha **Build**;



C# - Módulo I

- No projeto TestaClasses, crie um método para o evento Click do botão btnRetangulo.

```
private void btnRetangulo_Click(object sender, EventArgs e)
{
    // abre bloco de instruções protegidas de erro
    try
    {
        // criar objeto da classe Retangulo
        Retangulo r = new Retangulo();
        // atribuir valor às propriedades
        r.Base = updBase.Value;
        r.Altura = updAltura.Value;
        // executar os métodos para calcular a área e o perímetro
        lblAreaR.Text = r.Area().ToString();
        lblPerimetroR.Text = r.Perimetro().ToString();
    }
    // bloco de tratamento de erro
    // recebe o objeto erro ocorrido
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

11. Como não é possível calcular nada do retângulo sem informar sua base e altura, crie uma nova versão do construtor que receba a base e a altura como parâmetros. Você também pode criar um construtor para alterar o valor default de uma propriedade:

- Se for do tipo **int**, seu valor será 0 (zero);
- Se for do tipo **boolean**, seu valor será false;
- Se for do tipo **String**, seu valor será null.

```
// construtores  
// quando não criamos nenhum construtor, o compilador cria  
// um construtor padrão, sem parâmetros, mas quando criamos  
// um construtor alternativo, o compilador não cria um  
// construtor padrão  
public Retangulo()  
{  
    _base = 1;  
    _altura = 1;  
}  
  
public Retangulo(decimal b, decimal a)  
{  
    // para forçar a execução do método SET de cada  
    // propriedade, vamos atribuir os valores para a  
    // propriedade e não para a variável  
    Base = b;  
    Altura = a;  
}
```

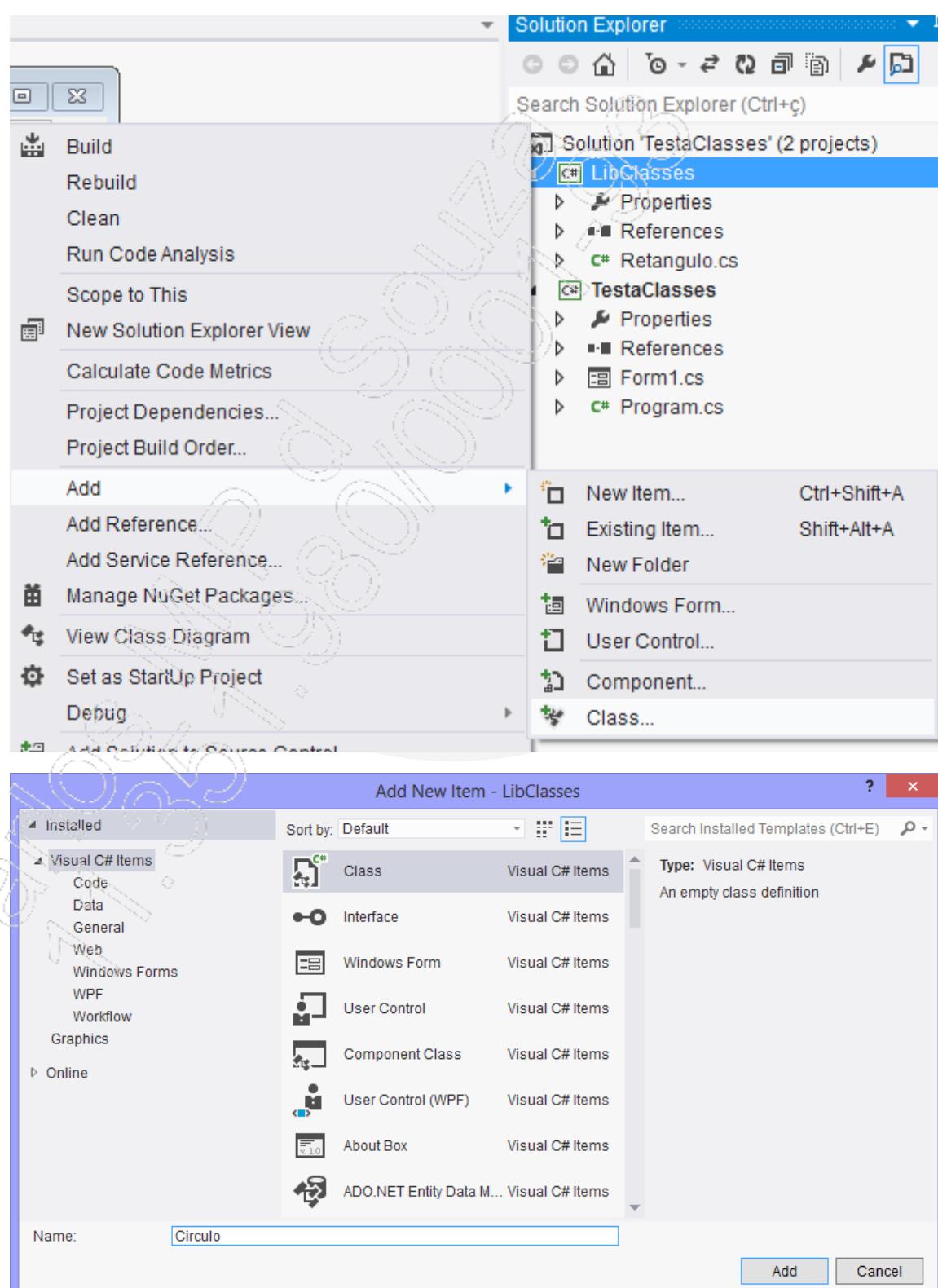
12. Compile novamente a biblioteca **LibClasses** e depois altere o evento **Click** do botão **btnRetangulo**:

```
private void btnRetangulo_Click(object sender, EventArgs e)  
{  
    // abre bloco de instruções protegidas de erro  
    try  
    {  
        // criar objeto da classe Retangulo  
        Retangulo r = new Retangulo(updBase.Value, updAltura.Value);  
        // executar os métodos para calcular a área e o perímetro  
        lblAreaR.Text = r.Area().ToString();  
        lblPerimetroR.Text = r.Perimetro().ToString();  
    }  
    // bloco de tratamento de erro  
    // recebe o objeto erro ocorrido  
    catch (Exception ex)  
    {  
        MessageBox.Show(ex.Message);  
    }  
}
```

9.4.2. Exemplo: Classe Circulo

Agora, criaremos uma classe para fazer cálculos geométricos envolvendo um círculo.

1. Em **LibClasses**, crie uma nova classe chamada **Circulo**. Para isso, no **Solution Explorer**, clique com o botão direito do mouse em **LibClasses**, selecione **Add** e escolha **Class**;



2. Crie uma propriedade **Raio** do tipo decimal. Ela não deve aceitar conteúdo zero ou negativo;

3. Crie um método para retornar a medida da área do círculo. O nome do método será **Area()** e deve retornar um valor do tipo decimal:

- **area = Math.PI * _raio * _raio**

! Como **Math.PI** foi definido como double e **_raio** como decimal, você precisará converter **Math.PI** para decimal também.

4. Crie um método para retornar a medida do perímetro do círculo. O nome do método será **Perímetro()** e deve retornar um valor do tipo decimal:

- **area = 2 * Math.PI * _raio**

! Como **Math.PI** foi definido como double e **_raio** como decimal, você precisará converter **Math.PI** para decimal também.

5. Crie os construtores indicados a seguir:

- Padrão (sem parâmetros) definindo valor inicial do raio igual a 1;
- Alternativo que vai receber o raio como parâmetro.

6. Compile a biblioteca e depois faça o evento **Click** do botão **btnCirculo** do projeto **TestaClasses**.

9.4.3. Exemplo: Classe ConvTemperaturas

A classe **ConvTemperaturas** deve fazer as conversões de medidas de temperatura entre as unidades Celsius, Fahrenheit e Kelvin.

1. Em **LibClasses**, crie uma nova classe chamada **ConvTemperaturas**. Esta classe terá três propriedades, todas do tipo decimal:

- Celsius;
- Fahrenheit;
- Kelvin.

2. Estabeleça as validações, definindo valores que as propriedades não podem aceitar:

- Celsius não pode ser menor que -273;
- Fahrenheit não pode ser menor que -459.4;
- Kelvin não pode ser menor que zero.

3. As conversões de unidade vão ocorrer da seguinte forma: sempre que atribuir valor a uma das propriedades, as outras duas devem ser recalculadas. Lembre-se de que quando atribuir valor a uma propriedade, a classe executará o método SET desta propriedade:

- Celsius

```
public decimal Celsius
{
    get { return _celsius; }
    set
    {
        if (_celsius >= -273)
        {
            _celsius = value;
            _fahrenheit = (_celsius * 9 / 5) + 32;
            _kelvin = _celsius + 273;
        }
        else
            throw new Exception("Não pode ser menor que -273");
    }
}
```

- Fahrenheit (complete)

```
_celsius = 5 * (_fahrenheit - 32) / 9;
_kelvin = _celsius + 273;
```

- Kelvin (complete)

```
_celsius = _kelvin - 273;
_fahrenheit = (_celsius * 9 / 5) + 32;
```

4. Crie o evento Click do botão que testa a classe ConvTemperaturas.

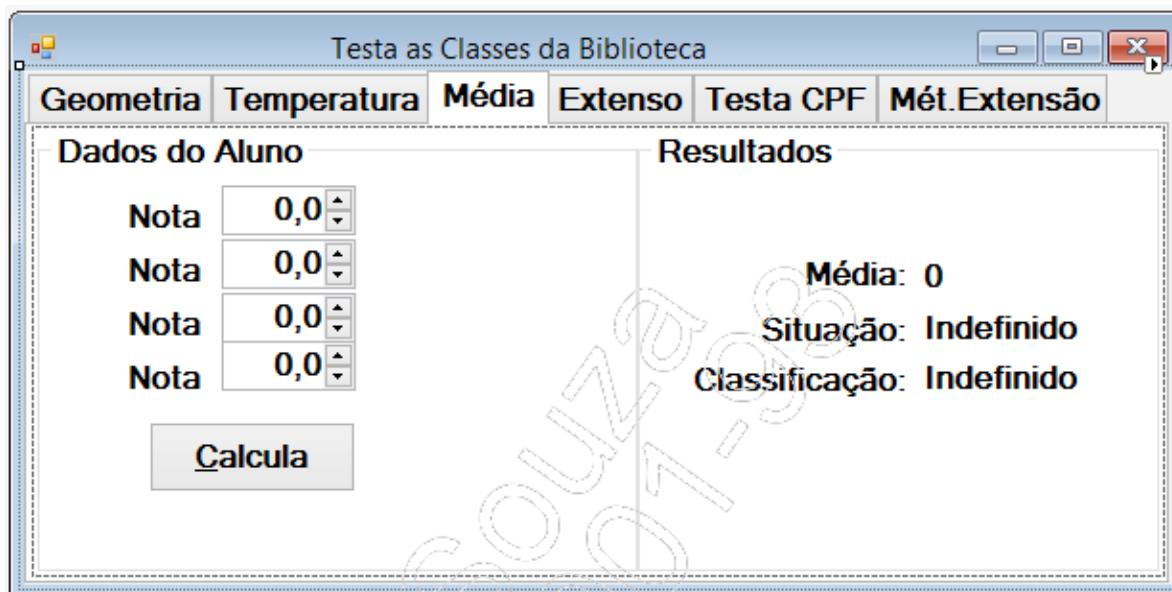
```
private void btnTemperatura_Click(object sender, EventArgs e)
{
    ConvTemperaturas ct = new ConvTemperaturas();

    if (rbCelsius.Checked) ct.Celsius = updTemp.Value;
    else if (rbFarenheit.Checked) ct.Fahrenheit = updTemp.Value;
    else ct.Kelvin = updTemp.Value;

    lblCelsius.Text = ct.Celsius.ToString("0.00");
    lblFarenheit.Text = ct.Fahrenheit.ToString("0.00");
    lblKelvin.Text = ct.Kelvin.ToString("0.00");
}
```

9.4.4. Exemplo: Classe CalcMedia

No início do curso, criamos um projeto chamado **Média**, que calculava a média das notas dos alunos e informava sua situação e sua classificação. Agora vamos recriá-lo, mas com toda a solução do problema dentro de uma classe. O projeto **TestaClasses** tem uma aba que faz o mesmo cálculo da média:



Observe o evento **Click** do botão que calcula a média:

```
private void btnCalcula_Click(object sender, EventArgs e)
{
    // bloco protegido de erro
    try
    {
        // cria objeto da classe CalcMedia passando para o construtor
        // as quatro notas
        CalcMedia cm = new CalcMedia(updN1.Value, updN2.Value,
                                     updN3.Value, updN4.Value);
        // Mostra o valor da média, situação e a classificação
        // utilizando a propriedades Read-Only que se autocalculam
        // usando o método get
        lblMedia.Text = cm.Media.ToString();
        lblSit.Text = cm.Situacao;
        lblClass.Text = cm.Classificacao;
    }
    // bloco de tratamento de erro
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

1. Em LibClasses, crie a classe **CalcMedia**;

2. Torne a classe **public** e crie uma propriedade para cada uma das 4 notas. A classe não deve aceitar notas fora do intervalo de 0 até 10;

```
namespace LibClasses
{
    public class CalcMedia
    {
        // criar as propriedades Nota1, Nota2, Nota3 e Nota4
        // todas do tipo decimal
        private decimal _nota1; // CTRL R+E

        public decimal Nota1
        {
            get { return _nota1; }
            set
            {
                if (value >= 0 && value <= 10) _nota1 = value;
                else throw new Exception("Nota 1 fora do intervalo");
            }
        }
    }
}
```

3. Complete criando as outras três propriedades para as notas;

4. Crie a propriedade **Media**, que deve ser read-only, e o seu método **set** retorna com o resultado do cálculo da média;

```
// propriedades de resultado -----
public decimal Media
{
    get { return (_nota1 + _nota2 + _nota3 + _nota4) / 4; }
    // não tem método SET porque esta propriedade não
    // pode receber valor, só pode ser consultada
}
```

5. Crie a propriedade **Situacao** (string) que testa, no seu método get. Se a média for menor que 5, retorna **REPROVADO**, caso contrário, retorna **APROVADO**. Esta propriedade não terá método set porque não pode receber valor;

6. Crie a propriedade **Classificacao** (string), que não tem método set e cujo método get retorna com uma das mensagens a seguir:

Média	Classificação
Até 2	Péssimo
Acima de 2 até 4	Ruim
Acima de 4 até 6	Regular
Acima de 6 até 8	Bom
Acima de 8	Ótimo

7. Crie um construtor padrão, que não recebe parâmetros;
8. Crie um construtor alternativo que receba as quatro notas como parâmetro;
9. Recompile a biblioteca e teste.

9.5. Classes static

As classes **static** funcionam como recipientes de campos e métodos utilitários. Uma classe marcada como **static** só pode ter membros estáticos. Seu uso é indicado nas situações em que desejamos evitar a associação de métodos a um objeto específico. Podemos utilizá-las no caso de não haver comportamento ou dados na classe que dependam da identidade do objeto.

Uma das principais características das classes estáticas é que elas são seladas, ou seja, não podem ser herdadas.

Quando o programa ou o namespace que possui a classe estática é carregado, o CLR (Common Language Runtime) do .NET Framework carrega automaticamente as classes estáticas.

O trabalho de criar uma classe estática é muito semelhante à criação de uma classe que possua somente membros estáticos e um construtor privado – o qual evita o instanciamento da classe. Não podemos utilizar a palavra-chave **new** para criar instâncias de uma classe estática.

Além de não poderem ser instanciadas, as classes estáticas não podem ter **Instance Constructors**. Embora possamos declarar um construtor estático para atribuir valores iniciais ou aplicar algum estado estático, a existência de construtores em classes estáticas não pode ocorrer.

! O compilador, além de não permitir a criação de instâncias de classes estáticas, verifica se membros de instância não foram acrescentados por acidente.

Vejamos o seguinte exemplo:

```
static class Geral
{
    //Método Estático
    public static string RetirarAcentos(string texto)
    {
        //Implementação do código do método
    }
}
```

Na linguagem C#, todas as tarefas são executadas dentro de classes, não é possível executar ação alguma fora de uma classe. No entanto, em algumas situações, o problema é tão simples que criar uma classe e depois instanciá-la para poder utilizá-la seria muito trabalhoso. Imagine que precisamos criar um recurso reutilizável para verificar se um CPF ou CNPJ é válido ou não, ou então gerar o extenso de um número. Estes são procedimentos independentes um do outro, que poderiam estar todos dentro de uma classe **static**.

Outra utilidade desse tipo de classe ou propriedade é manter determinados dados acessíveis a todos os módulos da aplicação, substituindo as antigas variáveis globais.

C# - Módulo I

```
namespace WindowsFormsApplication6
{
    public partial class Form1 : Form
    {
        public static string DadoGlobal = "Testando Dado Global";

        public Form1()
        {
            InitializeComponent();
        }
    }
}

namespace WindowsFormsApplication6
{
    public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
            // acessa a propriedade static de Form1
            label1.Text = Form1.DadoGlobal;
        }
    }
}
```

Estas são as principais características das classes **static**:

- Não pode gerar várias instâncias;
- Ela é carregada assim que o aplicativo que a utiliza é iniciado;
- Todos os seus membros também precisam ser **static**;
- É visível em todas as classes existentes no aplicativo que a utiliza;
- Seus membros são acessados diretamente pelo nome da classe:
 - **NomeClasse.NomePropriedade**;
 - **NomeClasse.NomeMetodo()**.

9.5.1. Modificador static

Podemos criar métodos e propriedades que podem ser acessados sem a necessidade de criar uma instância da classe. Para isso, adicionamos a palavra-chave **static** à esquerda do nome da classe ou membro da classe, gerando, assim, classes estáticas e membros estáticos.

9.5.1.1. Membros estáticos

Os membros estáticos são campos, métodos, eventos ou propriedades que podem ser chamados em uma classe mesmo que não tenha sido criada uma instância dessa classe.

Podemos utilizar membros estáticos para separar os dados e o comportamento que independem de qualquer identidade do objeto, ou seja, os dados e as funções não se modificam, não importa o que ocorra com o objeto.

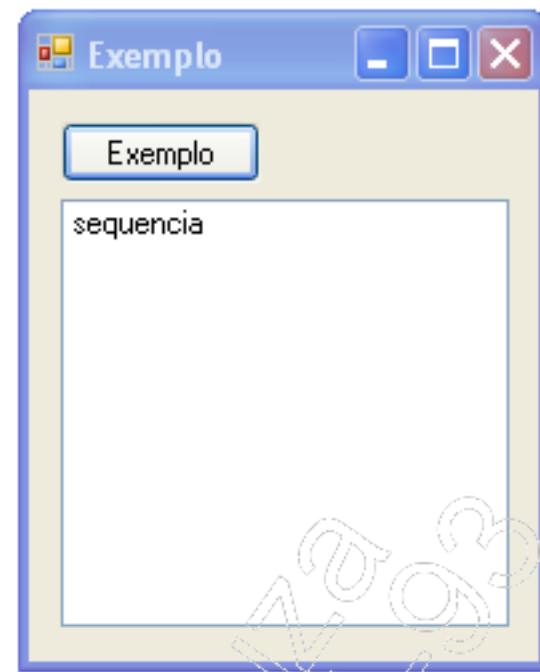
Vale destacarmos algumas considerações sobre a utilização de membros estáticos:

- Não podem ser acessados por meio de instâncias criadas para a classe;
- Propriedades e métodos estáticos podem acessar apenas campos estáticos e eventos estáticos;
- Eventos e campos estáticos possuem apenas uma cópia.

Vejamos um exemplo de utilização de métodos estáticos:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    resultadoListBox.Items.Add(Geral.RetirarAcentos("seqüência"));
}
```

O resultado desse código é o seguinte:



9.5.2. Exemplo: Classe Utils

Neste exemplo, vamos criar métodos para verificar a validade de CNPJ e CPF:

1. Crie uma nova classe chamada **Utils** no projeto **LibClasses**. Altere-a como mostra o código a seguir:

```
namespace LibClasses
{
    public static class Utils
    {
    }
}
```

2. O primeiro método que você criará tem como finalidade eliminar todos os caracteres não numéricos de uma string;

```
public static class Utils
```

```
{
    /// <summary>
    /// Elimina todos os caracteres não numéricos de uma string
    /// </summary>
    /// <param name="s">String de entrada</param>
    /// <returns>String somente com caracteres numéricos</returns>
    public static string Unformat(string s)
    {
        string r = "";
        // percorre cada um dos caracteres da string
        for (int i = 0; i < s.Length; i++)
        {
            // se for numérico concatena na variável de retorno
            if ("0123456789".Contains(s[i])) r += s[i];
        }
        return r;
    }
}
```

3. Crie o método para verificar se o CNPJ é válido ou não:

```

public static bool CNPJValido(string cnpj)
{
    int[] pesos1 = new int[12] { 5, 4, 3, 2, 9, 8, 7, 6, 5, 4, 3, 2 };
    int[] pesos2 = new int[13] { 6, 5, 4, 3, 2, 9, 8, 7, 6, 5, 4, 3, 2 };
    int soma;
    int resto;
    string digito;
    string temp;

    cnpj = Unformat(cnpj);

    if (cnpj.Length != 14)
        return false;

    temp = cnpj.Substring(0, 12);

    soma = 0;
    for (int i = 0; i < 12; i++)
        soma += Convert.ToInt32(temp[i].ToString()) * pesos1[i];

    resto = (soma % 11);
    resto = (resto < 2) ? 0 : 11 - resto;
}
```

C# - Módulo I

```
    digito = resto.ToString();

    temp = temp + digito;
    soma = 0;
    for (int i = 0; i < 13; i++)
        soma += Convert.ToInt32(temp[i].ToString()) * pesos2[i];

    resto = (soma % 11);

    resto = (resto < 2) ? 0 : 11 - resto;

    digito += resto.ToString();

    return cnpj.EndsWith(digito);
}
```

4. Crie o método para verificar se o CPF é válido:

```
public static bool CPFValido(string cpf)
{
    int n1, n2, n3, n4, n5, n6, n7, n8, n9;
    int d1, d2;
    string digitado, calculado;
    cpf = Unformat(cpf);

    if (cpf == "" || cpf.Length < 11)
    {
        return false;
    }
    n1 = Convert.ToInt32((cpf[0].ToString()));
    n2 = Convert.ToInt32((cpf[1].ToString()));
    n3 = Convert.ToInt32((cpf[2].ToString()));
    n4 = Convert.ToInt32((cpf[3].ToString()));
    n5 = Convert.ToInt32((cpf[4].ToString()));
    n6 = Convert.ToInt32((cpf[5].ToString()));
    n7 = Convert.ToInt32((cpf[6].ToString()));
    n8 = Convert.ToInt32((cpf[7].ToString()));
    n9 = Convert.ToInt32((cpf[8].ToString()));
    d1 = n9 * 2 + n8 * 3 + n7 * 4 + n6 * 5 + n5 * 6 + n4 * 7 + n3 * 8 +
         n2 * 9 + n1 * 10;
    d1 = 11 - (d1 % 11);
    if (d1 >= 10) d1 = 0;
```

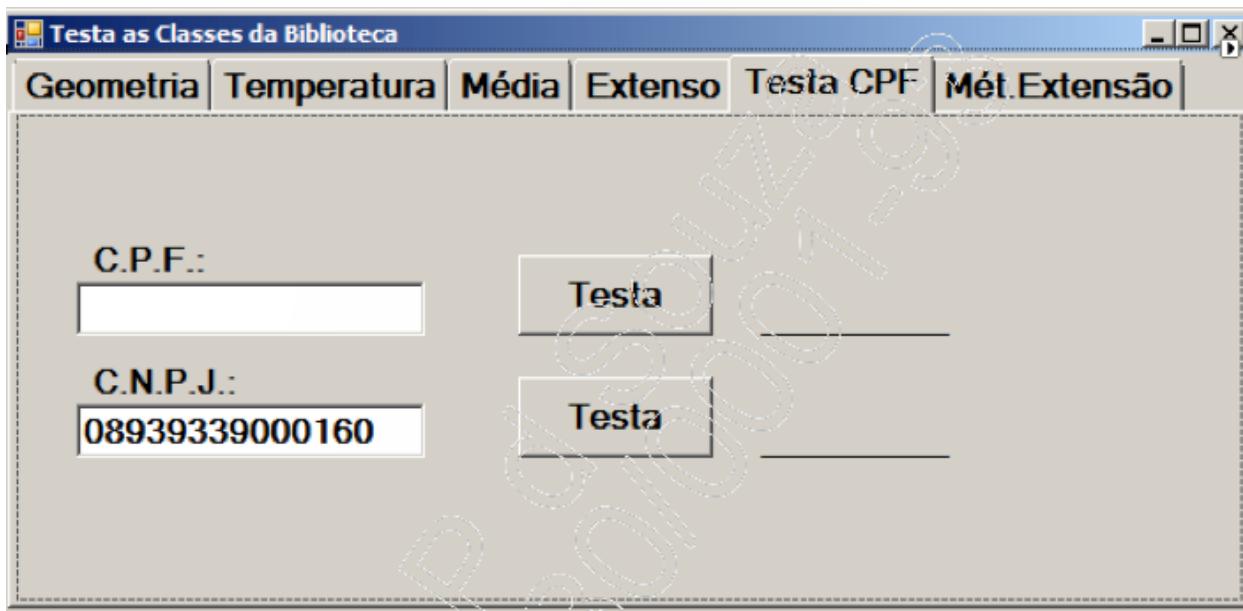
```

d2 = d1 * 2 + n9 * 3 + n8 * 4 + n7 * 5 + n6 * 6 + n5 * 7 + n4 * 8 +
     n3 * 9 + n2 * 10 + n1 * 11;
d2 = 11 - (d2 % 11);
if (d2 >= 10) d2 = 0;
calculado = d1.ToString() + d2.ToString();

digitado = cpf.Substring(9, 2);
return (calculado == digitado);
}

```

5. Recompile **LibClasses** e teste-o no projeto **TestaClasses**.



```

// evento Click do botão que testa o CPF
private void btnTestaCPF_Click(object sender, EventArgs e)
{
    if (Utils.CPFValido(tbxCPF.Text))
        lblCPF.Text = "Válido";
    else
        lblCPF.Text = "Inválido";
}
// evento Click do botão que testa o CNPJ
private void btnTestaCNPJ_Click(object sender, EventArgs e)
{
    if (Utils.CNPJValido(tbxCNPJ.Text))
        lblCNPJ.Text = "Válido";
    else
        lblCNPJ.Text = "Inválido";
}

```

9.6. Métodos de extensão

Este recurso permite a criação de novos métodos para uma classe já existente. Neste caso, não estamos alterando a classe original ou criando um herdeiro dela, estamos apenas disponibilizando métodos adicionais para uma aplicação específica.

Enquanto a herança permite trabalhar apenas com classes, os métodos de extensão possibilitam estender classes e estruturas também.

Ao utilizarmos um método de extensão, poderemos ampliar uma classe ou uma estrutura já existente com métodos estáticos adicionais. Dessa forma, os métodos estáticos podem ser utilizados ao longo do código, seja qual for a instrução que faça referência aos dados do tipo estendido.

The diagram illustrates an extension method definition and its use in a code snippet. The method definition is:

```
public static bool EhDecimal(this string valor)
```

The usage of the method is shown in a conditional statement:

```
if (tbxNumero.Text.EhDecimal())
    lblTesta.Text = "É numérico";
else
    lblTesta.Text = "NÃO É numérico";
```

A pink arrow points from the word 'valor' in the method signature to the 'valor' parameter in the if-block's condition, indicating they refer to the same variable.

Para definir um método de extensão, devemos usar uma classe estática. Para especificar o tipo que será aplicado à classe, basta defini-lo como o primeiro parâmetro para o método junto com a palavra-chave **this**, a qual caracteriza o método de extensão. Vejamos os itens indicados no código anterior:

- A - Modificador de acesso;
- B - O método de extensão tem que ser **static**;
- C - Tipo de retorno do método;

- **D** - Nome do método;
- **E** - Caracteriza um método de extensão;
- **F** - Classe ou estrutura para a qual estamos criando o método;
- **G** - Dado a ser manipulado pelo método. Note que, quando usamos o método, este valor não é passado explicitamente entre parênteses, o valor corresponde ao conteúdo do objeto que invoca o método.

9.6.1. Exemplo: Classe MetodosExtensao

Neste exemplo, vamos trabalhar com métodos de extensão:

1. Em **LibClasses**, crie uma nova classe chamada **MetodosExtensao**. Torne-a **public** e **static**. A classe contendo os métodos de extensão precisa ser **static**, bem como todos os seus métodos;

```
namespace LibClasses
{
    public static class MetodosExtensao
    {
    }
}
```

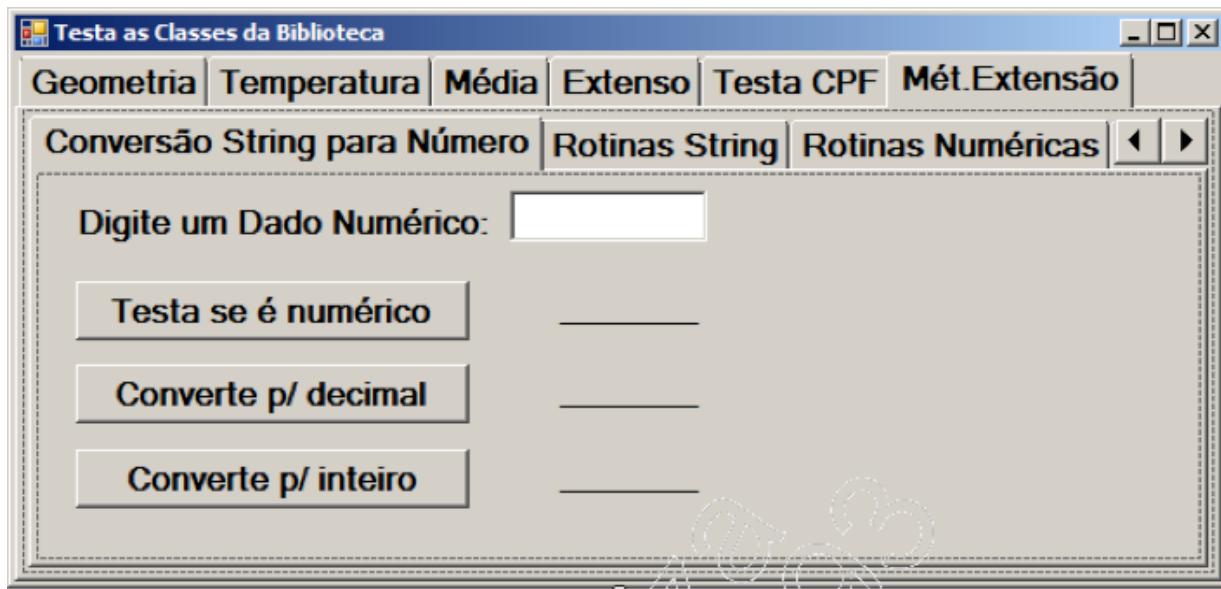
2. Crie os métodos indicados a seguir:

```
public static class MetodosExtensao
{
    ///<summary>
    /// Testa se um string pode ser convertido em decimal
    /// </summary>
    /// <param name="valor">String para ser testado</param>
    /// <returns>true se puder ser convertido ou false se não</returns>
    public static bool EhDecimal(this string valor)
```

C# - Módulo I

```
{  
    try  
    {  
        Convert.ToDecimal(valor);  
        return true;  
    }  
    catch  
    {  
        return false;  
    }  
}  
/// <summary>  
/// Converte string para decimal  
/// </summary>  
/// <param name="valor">string que será convertido</param>  
/// <returns>numero decimal</returns>  
public static decimal ToDecimal(this string valor)  
{  
    try  
    {  
        return Convert.ToDecimal(valor);  
    }  
    catch  
    {  
        throw new Exception("Dado não é numérico");  
    }  
}  
/// <summary>  
/// Converte o string para Int32  
/// </summary>  
/// <param name="valor">string que será convertido</param>  
/// <returns>Valor Int32 correspondente</returns>  
public static intToInt32(this string valor)  
{  
    // complete você  
  
}
```

3. Teste até este ponto. Compile **LibClasses** e crie o evento **Click** dos três botões presentes na janela a seguir:



```
private void btnTesta_Click(object sender, EventArgs e)
{
    if (tbxNumero.Text.EhDecimal())
        lblTesta.Text = "É numérico";
    else
        lblTesta.Text = "NÃO É numérico";
}

private void btnConvDec_Click(object sender, EventArgs e)
{
    try
    {
        lblConvDec.Text = tbxNumero.Text.ToDecimal().ToString();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

private void btnConvInt_Click(object sender, EventArgs e)
{
    // complete
}
```

4. Mais métodos de extensão para string. Inclua-os na classe **MetodosExtensao**:

```
/// <summary>
/// Conta quantas palavras tem um string usando como separador de palavra
/// qualquer um desses caracteres: ' ', ',', '.', '?', '!', ';' 
/// </summary>
/// <param name="texto">Texto</param>
/// <returns>Quantidade de palavras</returns>
public static int ContaPalavras(this string texto)
{
    string[] palavras =
        texto.Split(new char[] { ' ', ',', '.', '?', '!', ';' });
    return palavras.Length;
}

/// <summary>
/// Retorna com a palavra que está na posição pos do texto
/// </summary>
/// <param name="texto">Texto</param>
/// <param name="pos">Posição da palavra que queremos pegar</param>
/// <returns>Palavra que está na posição</returns>
public static string PegaPalavra(this string texto, uint pos)
{
    string[] palavras =
        texto.Split(new char[] { ' ', ',', '.', '?', '!', ';' });
    if (pos < palavras.Length)
        return palavras[pos];
    else
        return "";
}

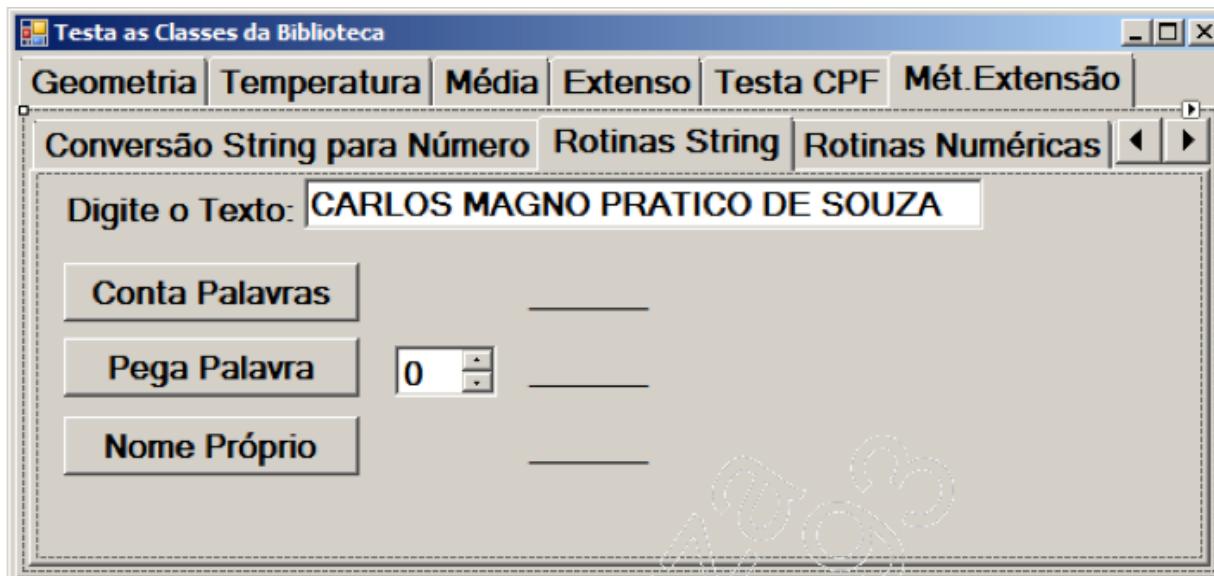
/// <summary>
/// Converte a primeira letra de cada palavra para maiúsculo
/// e as outras para minúsculo
/// </summary>
/// <param name="nome">Nome que será convertido</param>
/// <returns></returns>
public static string NomeProprio(this string nome)
{
```

```
string ret = "";
int qtdPalavras = nome.ContaPalavras();
for (int i = 0; i < qtdPalavras; i++)
{
    string p = nome.PegaPalavra((uint)i);
    if ("DE,DO,DAS,DOS".Contains(p.ToUpper()))
        p = p.ToLower();
    else
        p = p.Substring(0, 1).ToUpper() + p.Substring(1).ToLower();
    ret += p + " ";
}
return ret.Trim();
```

```
/// <summary>
/// Substitui os caracteres acentuados pelo seu correspondente
/// não acentuado
/// </summary>
/// <param name="texto">Texto com acentos</param>
/// <returns>Texto sem acentos</returns>
public static string TiraAcento(this string texto)
{
    string result = "";
    string acentos = "ÀÂÃáàâãÉÈÊËéèêëÍÌÏÏíïóòôõöôõöÚÙÛÜúùûüçÇ";
    string semAcentos = "AAAAaaaaEEEEeeeeIIIiili00000oooooUUUUuuuuucC";
    for (int i = 0; i < texto.Length; i++)
    {
        int pos = acentos.IndexOf(texto[i]);
        if (pos >= 0) result += semAcentos[pos];
        else result += texto[i];
    }
    return result;
}
```

C# - Módulo I

5. Teste até este ponto. Compile **LibClasses** e crie o evento **Click** dos três botões presentes na imagem a seguir:



```
// botão Conta Palavras
private void btnContaPal_Click(object sender, EventArgs e)
{
    lblContaPal.Text = tbxTexto.Text.ContaPalavras().ToString();
}
// botão Pega Palavra
private void btnPegaPal_Click(object sender, EventArgs e)
{
    lblPegaPal.Text = tbxTexto.Text.PegaPalavra((uint)updPos.Value);
}
// botão Nome Próprio
private void btnNomeProprio_Click(object sender, EventArgs e)
{
    lblNomeProprio.Text = tbxTexto.Text.NomeProprio();
```

9.7. Herança

A herança possibilita que as classes compartilhem atributos, métodos e outros membros de classe entre si. Para a ligação entre as classes, a herança adota um relacionamento esquematizado hierarquicamente. Na herança, temos dois tipos principais de classe:

- **Classe base:** A classe que concede as características a outra classe;
- **Classe derivada:** A classe que herda as características da classe base.

O fato de as classes derivadas herdarem atributos das classes base assegura que a complexidade de programas orientados a objetos cresça de forma linear, e não geométrica.

Uma nova classe derivada não possui interações imprevisíveis em relação ao restante do código do sistema. Com o uso da herança, uma classe derivada geralmente é uma implementação específica de um caso mais geral. A classe derivada deve apenas definir as características que a tornam única.

Quando trabalhamos com herança, pode ser interessante que alguns itens da classe fiquem acessíveis somente para as classes derivadas, mas não para outras classes que não fazem parte da hierarquia. Nesses casos, podemos usar a palavra-chave **protected**, a qual é considerada como um meio termo entre **public** e **private**. Ao definirmos os membros de uma classe como **protected**, apenas as classes derivadas dessa primeira terão acesso a esses membros, enquanto as classes que não forem derivadas dela não poderão acessá-los. Também terão acesso a membros protegidos as classes derivadas de classes também derivadas.

9.7.1. Criando uma herança

A seguinte sintaxe é utilizada na criação de herança:

```
class ClasseDerivada : ClasseBase
{
    ...
}
```

C# - Módulo I

O exemplo a seguir ilustra a criação efetiva de uma herança. Consideremos a classe **Funcionario**:

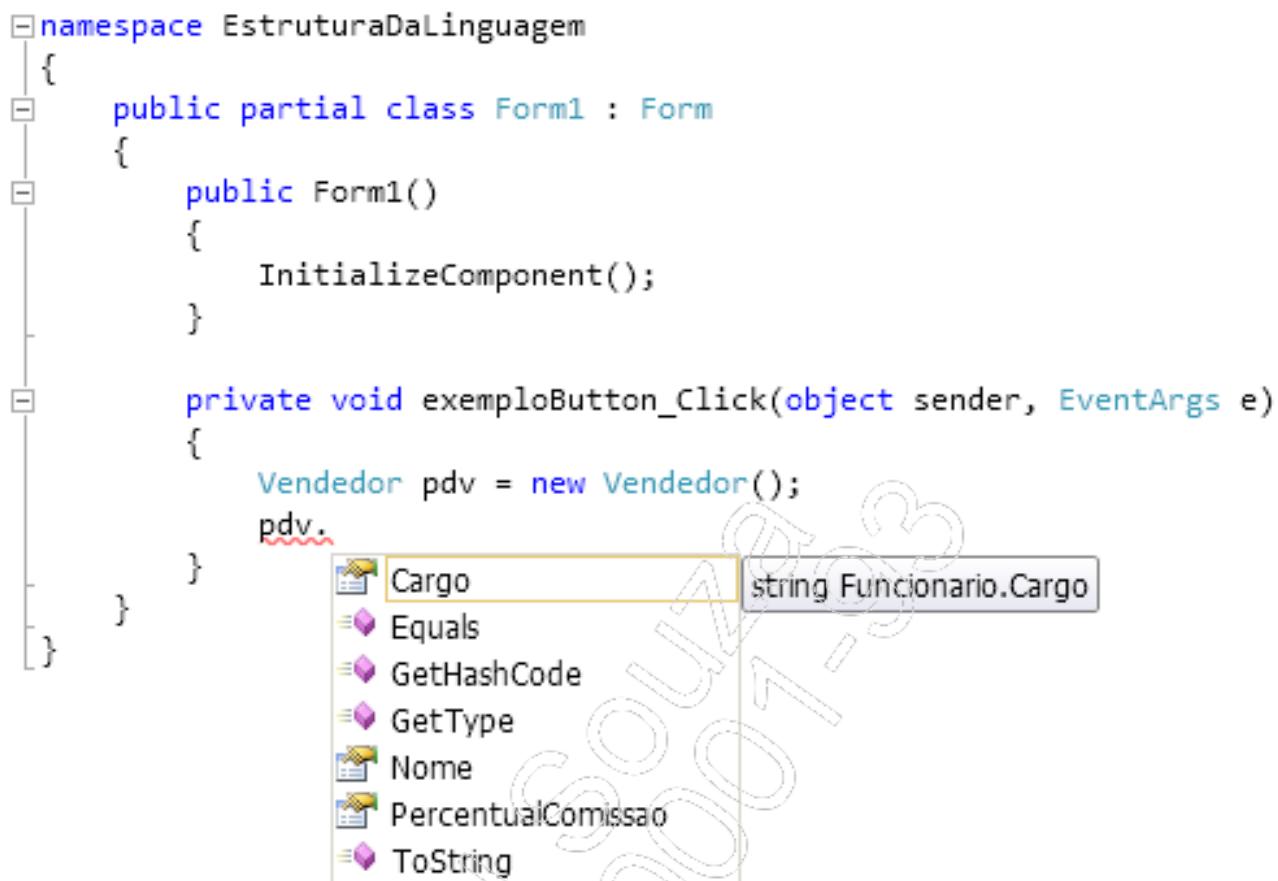
```
class Funcionario
{
    private string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }
    private string _cargo;
    public string Cargo
    {
        get { return _cargo; }
        set { _cargo = value; }
    }
}
```

Então, consideremos a criação de um cadastro de vendedores de uma empresa. Entendemos que cada vendedor é um funcionário, mas nem todo funcionário é um vendedor. Essa relação nos obriga a criar uma extensão da classe **Funcionario** a fim de que a mesma contenha os dados específicos referentes aos vendedores.

Desse modo, a extensão criada acaba herdando os membros existentes na definição da classe **Funcionario**. A seguir, temos a definição da classe **Vendedor**, que, pela herança, recebe as características da classe **Funcionario**, somadas à definição de seus próprios membros:

```
class Vendedor : Funcionario
{
    private float _percentualComissao;
    public float PercentualComissao
    {
        get { return _percentualComissao; }
        set { _percentualComissao = value; }
    }
}
```

O resultado desse código é o seguinte:



```
namespace EstruturaDaLinguagem
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void exemploButton_Click(object sender, EventArgs e)
        {
            Vendedor pdv = new Vendedor();
            pdv.
```

A screenshot of the Microsoft Visual Studio IDE showing code completion. The cursor is at the end of 'pdv.' and a dropdown menu lists several members of the 'Vendedor' class, including 'Cargo' (highlighted in yellow), 'Equals', 'GetHashCode', 'GetType', 'Nome', 'PercentualComissao', and 'ToString'. The 'Cargo' member is currently selected.

A classe **Vendedor** que acabamos de criar é uma extensão (também chamada de classe filha, derivada ou subclasse) da classe **Funcionario** (também chamada de classe pai ou superclasse).

Como vimos no exemplo, os dois pontos (:), que definem a herança, fazem com que a classe **Vendedor** herde as características da classe **Funcionario**.

É importante ressaltar que, ao contrário de algumas outras linguagens, o C# não trabalha com herança múltipla.

9.7.2. Acesso aos membros da classe pai por meio do operador base

Qualquer referência feita com uso do operador **base**, dentro de uma classe filha, será apontada diretamente para a classe pai. Apesar de não ser um procedimento recomendado, podemos manter atributos de nomes iguais nas classes pai e filha. Esse procedimento é chamado de **hiding**. Quando o utilizamos, devemos adicionar a instrução **new** à frente do atributo da classe derivada, caso contrário, o compilador irá enviar um alerta sobre o uso de **hiding**.

Vejamos um exemplo de código:

```
class VeiculoBase
{
    public string Fabricante = "Fiat";
}

class VeiculoCarga : VeiculoBase
{
    public new string Fabricante = "Mercedes-Benz";

    public string MostrarDados()
    {
        return
            "Classe Pai: " + base.Fabricante +
            "\nClasse Filha: " + this.Fabricante;
    }
}
```

O resultado desse código pode ser visto a seguir:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    VeiculoCarga caminhao = new VeiculoCarga();

    MessageBox.Show(caminhao.MostrarDados());
}
```



9.7.3. Métodos sobrecarregados (overloaded methods)

São métodos que possuem o mesmo nome, mas parâmetros diferentes.

Vejamos um exemplo:

```
class Automovel
{
    //Sobrecarga de método
    public string MostrarDados(string fabricante)
    {
        return "Fabricante: " + fabricante;
    }

    public string MostrarDados(string fabricante, string modelo)
    {
        return
            "Fabricante: " + fabricante +
            " - Modelo: " + modelo;
    }
}
```

C# - Módulo I

No formulário:

```
private void exemploButton_Click(object sender, EventArgs e)
{
    Automovel carro = new Automovel();

    exemploListBox.Items.Add(carro.MostrarDados("Fiat"));
    exemploListBox.Items.Add(carro.MostrarDados("Fiat", "Pálio"));
}
```

O resultado desse código é o seguinte:



A principal utilidade da sobrecarga é a possibilidade de fazer com que uma mesma operação seja executada em tipos de dados diferentes. Assim, quando temos implementações de mesmo nome, mas com um número diferente de parâmetros, ou parâmetros de tipos diferentes, podemos sobrepor um método. Dessa forma, torna-se possível definir uma lista de argumentos (os quais separamos por vírgulas) ao chamarmos um método e, baseado no número e tipo dos argumentos, o compilador escolherá um dos métodos sobreporados.

! Não é possível declarar dois métodos com o mesmo nome, mesmos parâmetros e tipos de retorno diferentes, ou seja, a sobreposição do tipo de retorno não é permitida.

9.7.4. Polimorfismo

Definimos polimorfismo como sendo um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de forma específica para cada uma das classes derivadas.

É importante destacar que a assinatura de um método diz respeito somente ao seu nome e aos tipos e números de seus parâmetros, aspectos esses que, caso sejam os mesmos em métodos diferentes, levarão esses métodos a terem a mesma assinatura, ainda que seus tipos de retorno possam ser diferentes. Isso só não se aplica quando o tipo de retorno determina a compatibilidade entre um delegate e o método que ele aponta.

Como exemplo de polimorfismo, consideremos uma classe chamada **Cliente** e outra chamada **Funcionario** que têm como base uma classe chamada **Pessoa** com um método chamado **EnviarEmail**.

Se esse método (definido na classe base) se comportar de maneira diferente, dependendo do fato de ter sido chamado a partir de uma instância de **Cliente** ou a partir de uma instância de **Funcionario**, será considerado um método polimórfico, ou seja, um método de várias formas. Esse princípio é chamado polimorfismo.

9.7.5. Palavras-chaves virtual e override

Com a utilização da palavra-chave **virtual** – destinada a métodos, propriedades, eventos e indexadores –, determinamos que um membro pode ser sobreescrito em uma classe filha. Por meio da palavra-chave **override**, determinamos que, na classe derivada, um membro virtual da classe base pode ser sobreescrito.

Agora que já descrevemos a utilidade das palavras-chave **virtual** e **override**, devemos atentar a dois aspectos importantes. Primeiramente, devemos saber que se trata de uma diferença do C# em comparação à linguagem Java. Esta não possui as palavras-chave em questão, sendo os métodos implicitamente virtuais. Além desse aspecto, devemos ter em mente que ambas as palavras-chave complementam uma à outra.

É importante saber ainda que a propagação da palavra-chave **virtual** ocorre para descendentes. Um método ou propriedade virtual pode ser sobreescrito em descendentes e até mesmo em uma classe derivada.

Existem algumas regras às quais devemos obedecer quando trabalhamos com **virtual** e **override**:

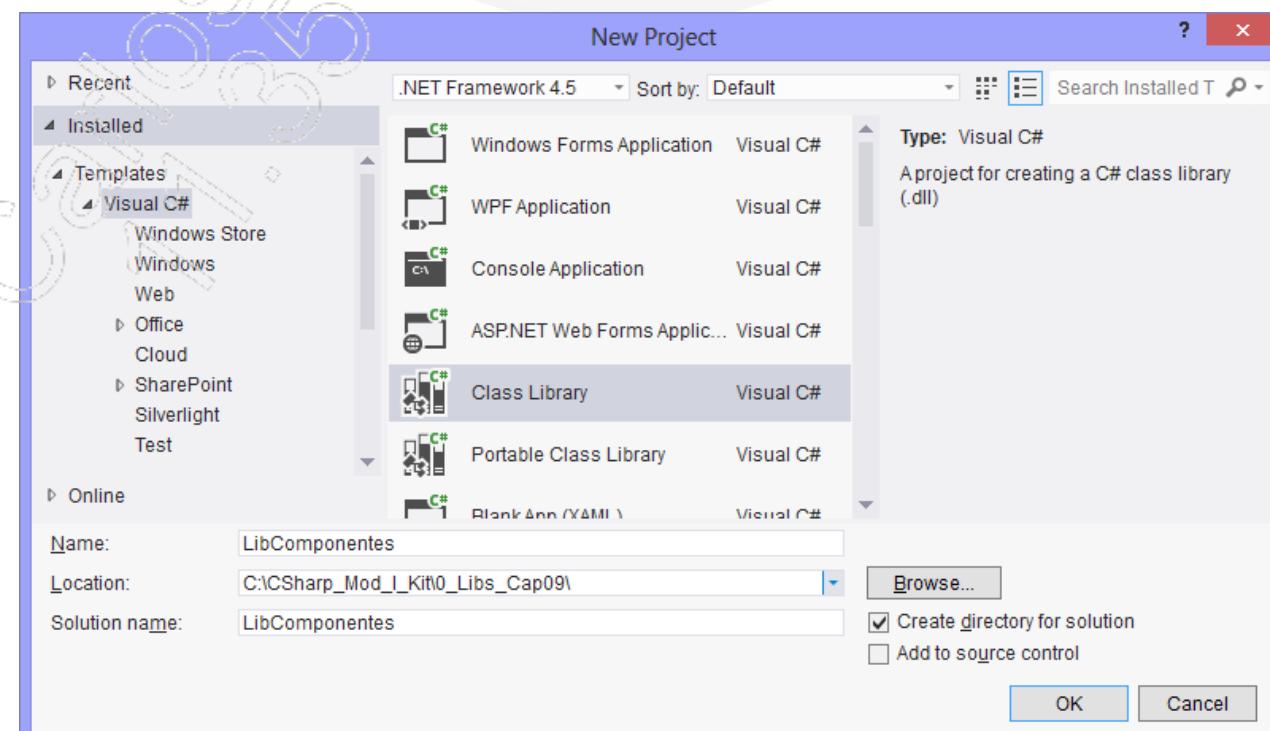
- Devemos nos certificar de que ambos os métodos têm o mesmo acesso, lembrando que um método privado não pode ser declarado com uma dessas palavras-chave;
- Quanto às assinaturas de método, as duas devem ser idênticas (mesmo nome e mesmos tipos e números de parâmetros);
- A redefinição só pode ser feita em métodos virtuais, o que é definido na classe base;
- É necessário que a classe derivada declare o método usando a palavra-chave **override** para que a redefinição do método da classe base de fato ocorra.

Não devemos declarar explicitamente um método **override** com a palavra-chave **virtual**; esse tipo de método já é implicitamente virtual, sendo permitida sua redefinição em uma classe derivada posterior.

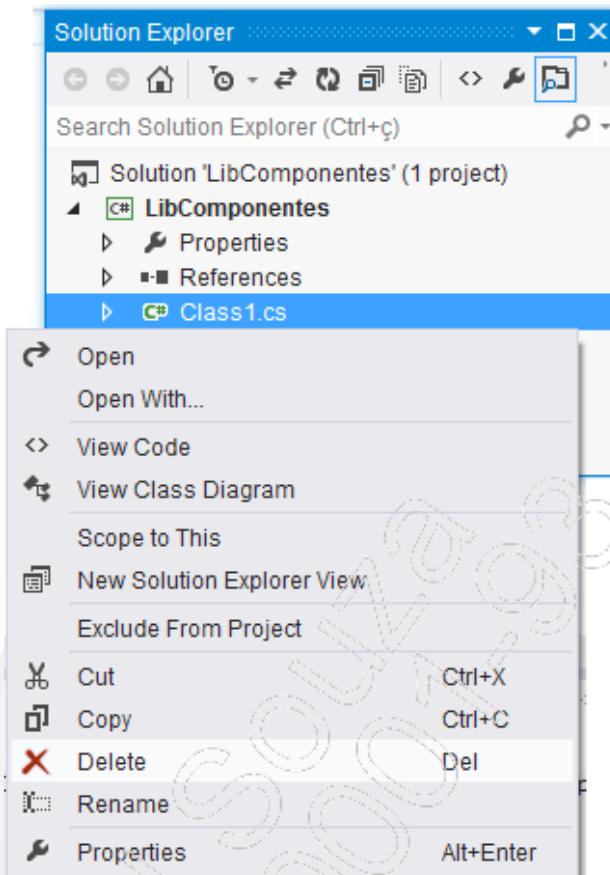
9.7.6. Exemplo: Class Library LibComponentes

Neste exemplo, vamos trabalhar com herança entre diversas classes.

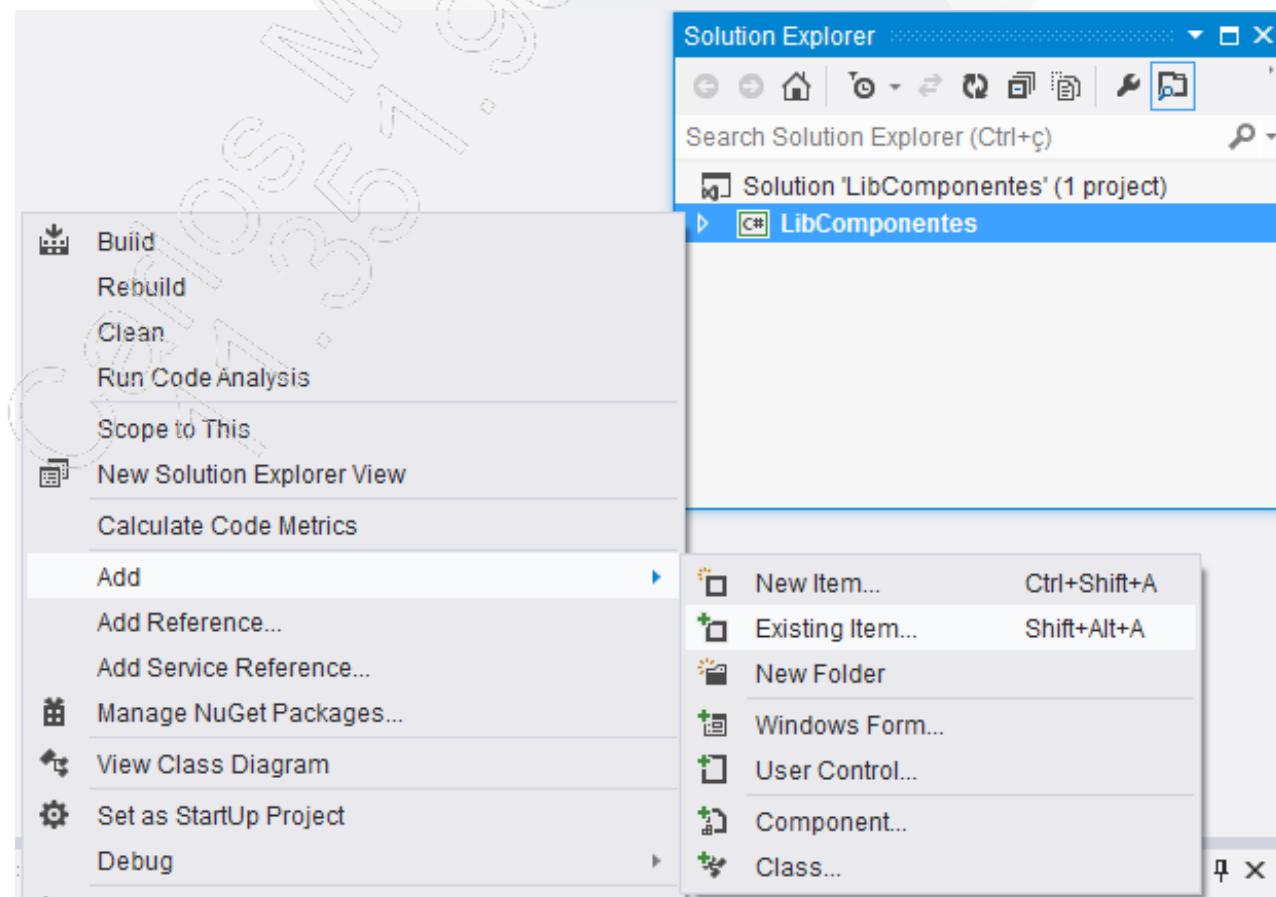
1. Feche todos os projetos e crie uma nova **Class Library** chamada **LibComponentes**. Selecione a pasta **0_Libs_Cap09** para gravar o projeto;



2. Apague o arquivo **Class1.cs** do projeto;

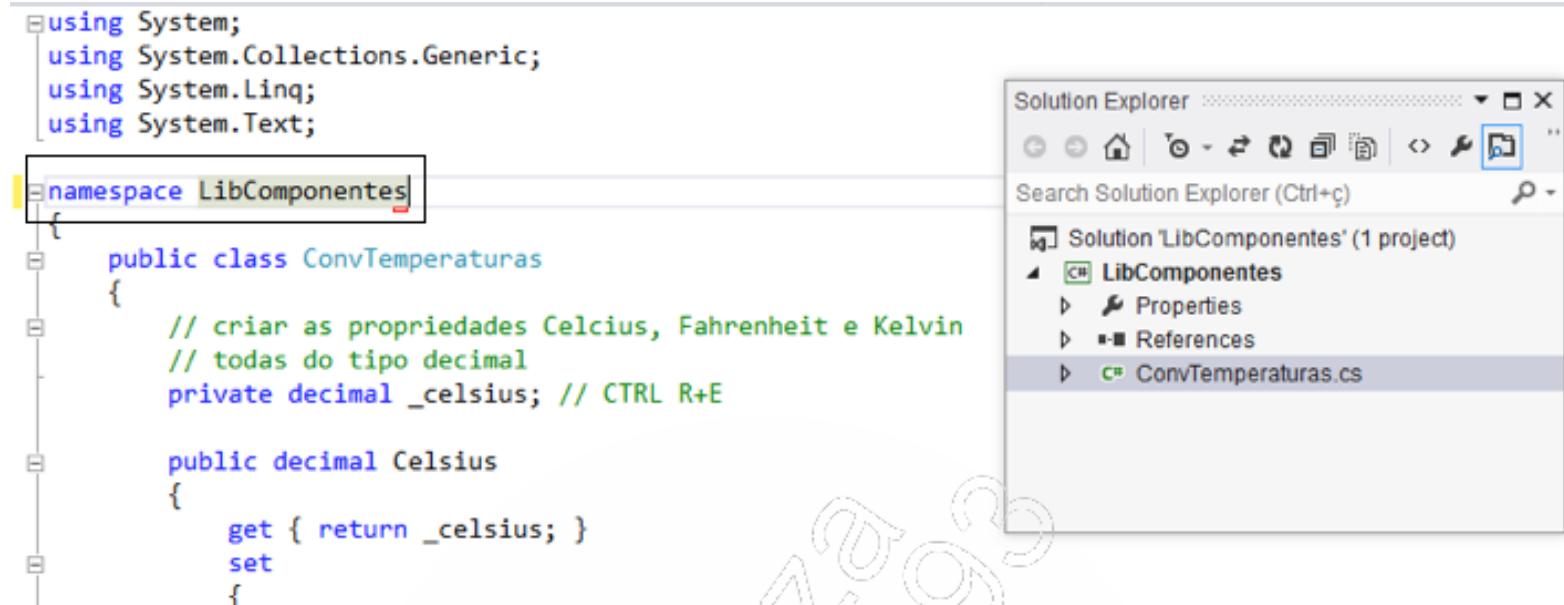


3. Adicione em **LibComponentes** o arquivo **ConvTemperaturas** criado em **LibClasses**;



C# - Módulo I

4. Altere o namespece de ConvTemperaturas para LibComponentes;



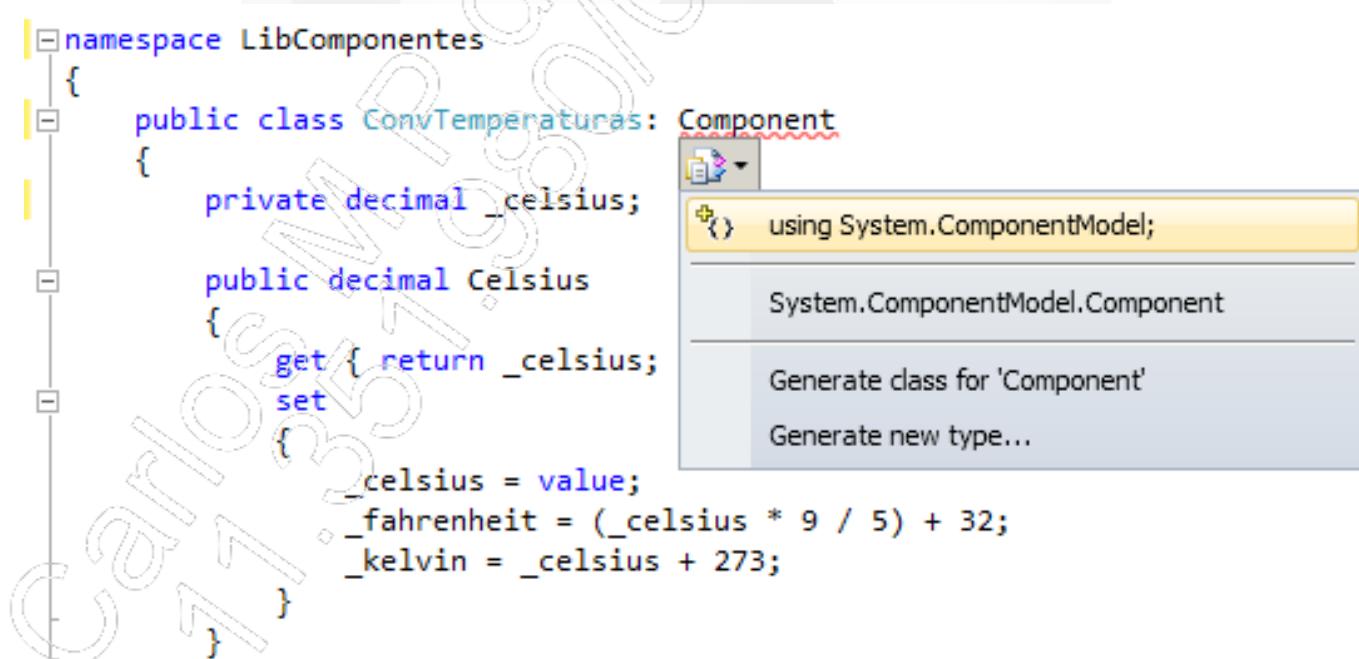
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LibComponentes
{
    public class ConvTemperaturas
    {
        // criar as propriedades Celcius, Fahrenheit e Kelvin
        // todas do tipo decimal
        private decimal _celsius; // CTRL R+E

        public decimal Celsius
        {
            get { return _celsius; }
            set
            {

```

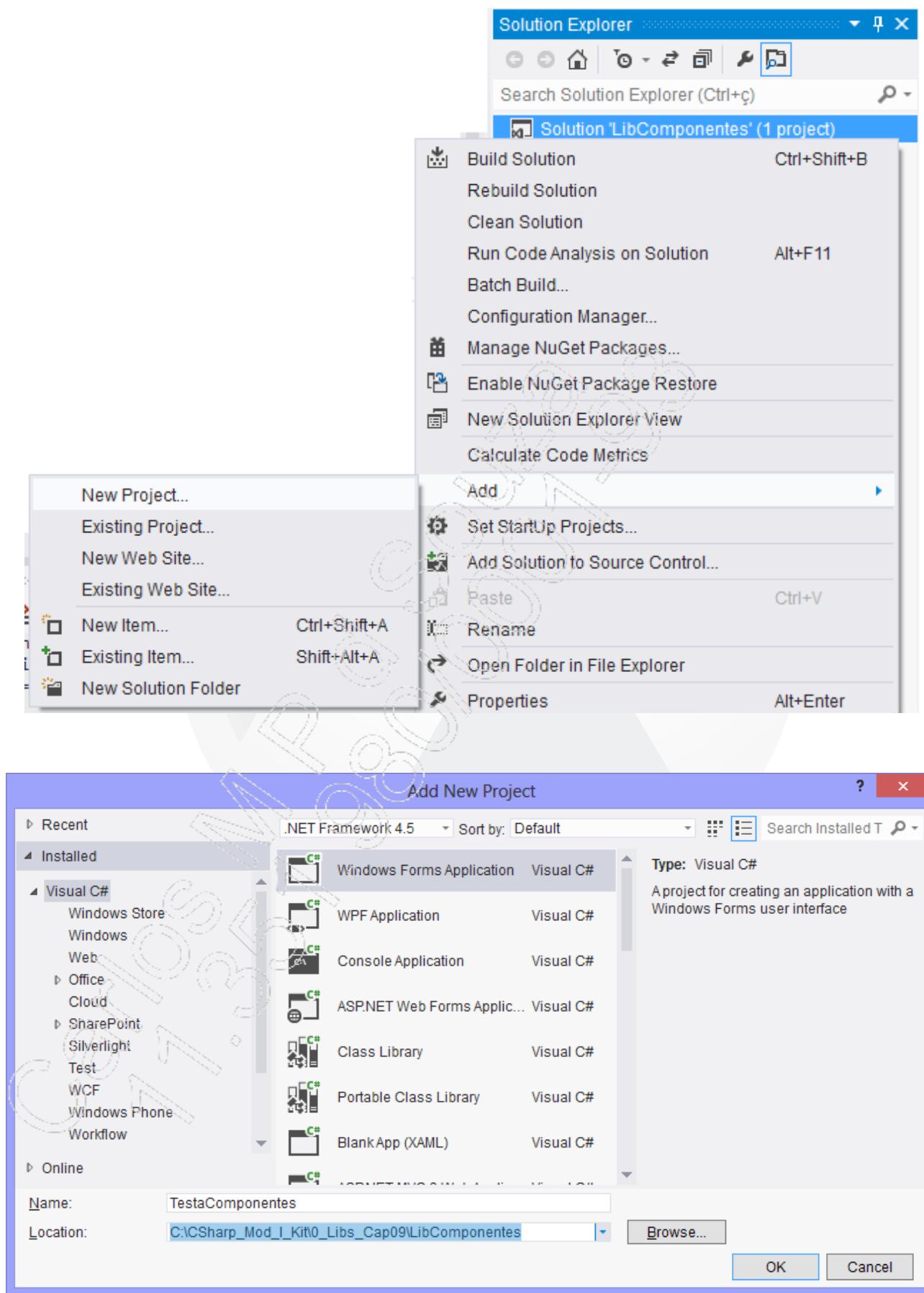
5. Altere a declaração da classe, indicando que ela é herdeira da classe **Component**. Os herdeiros da classe **Component** têm a possibilidade de aparecerem na ToolBox;



```
namespace LibComponentes
{
    public class ConvTemperaturas: Component
    {
        private decimal _celsius;

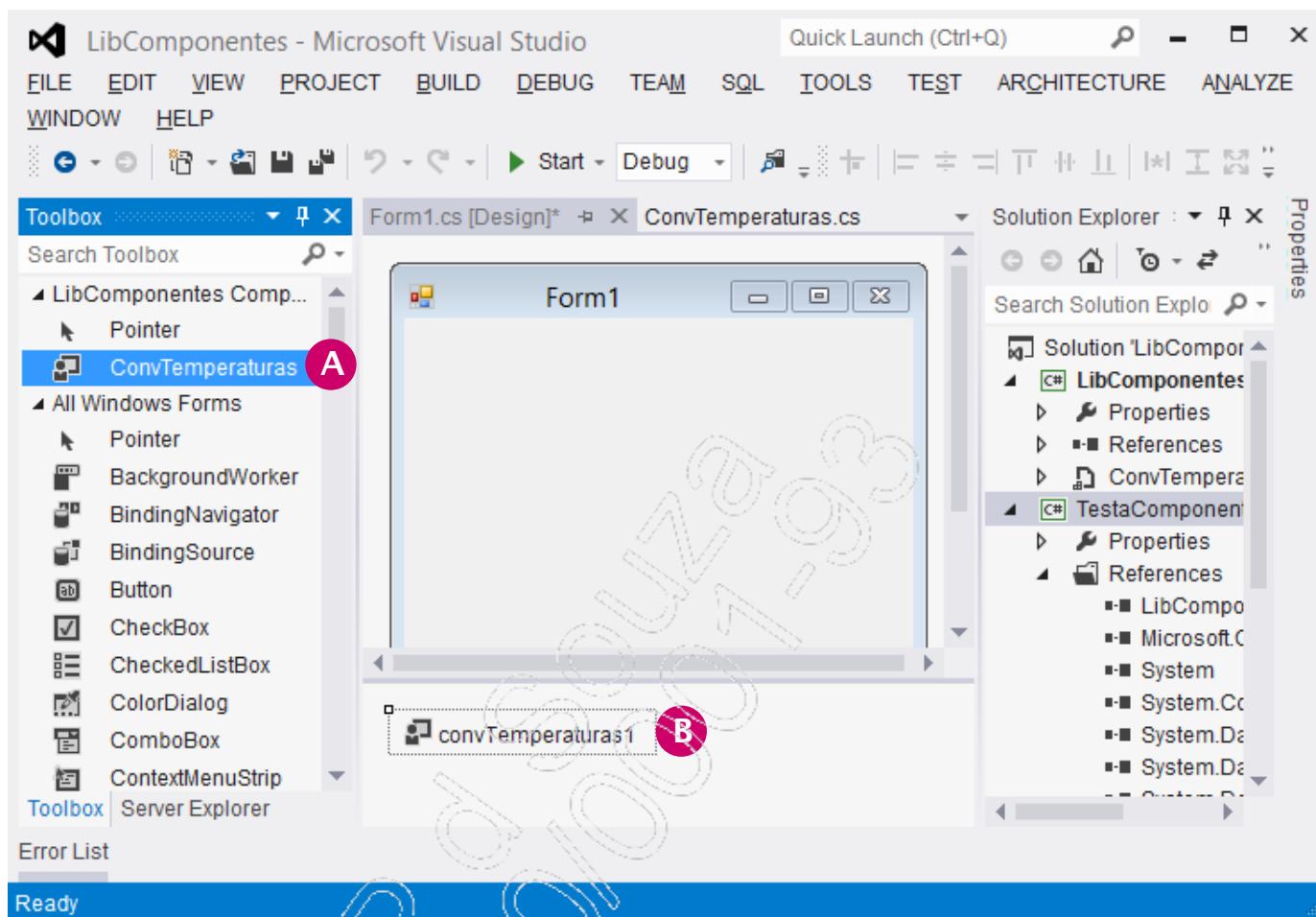
        public decimal Celsius
        {
            get { return _celsius; }
            set
            {
                _celsius = value;
                _fahrenheit = (_celsius * 9 / 5) + 32;
                _kelvin = _celsius + 273;
            }
        }
}
```

6. Compile LibComponentes e adicione na solução um novo projeto Windows Form chamado **TestaComponentes**;

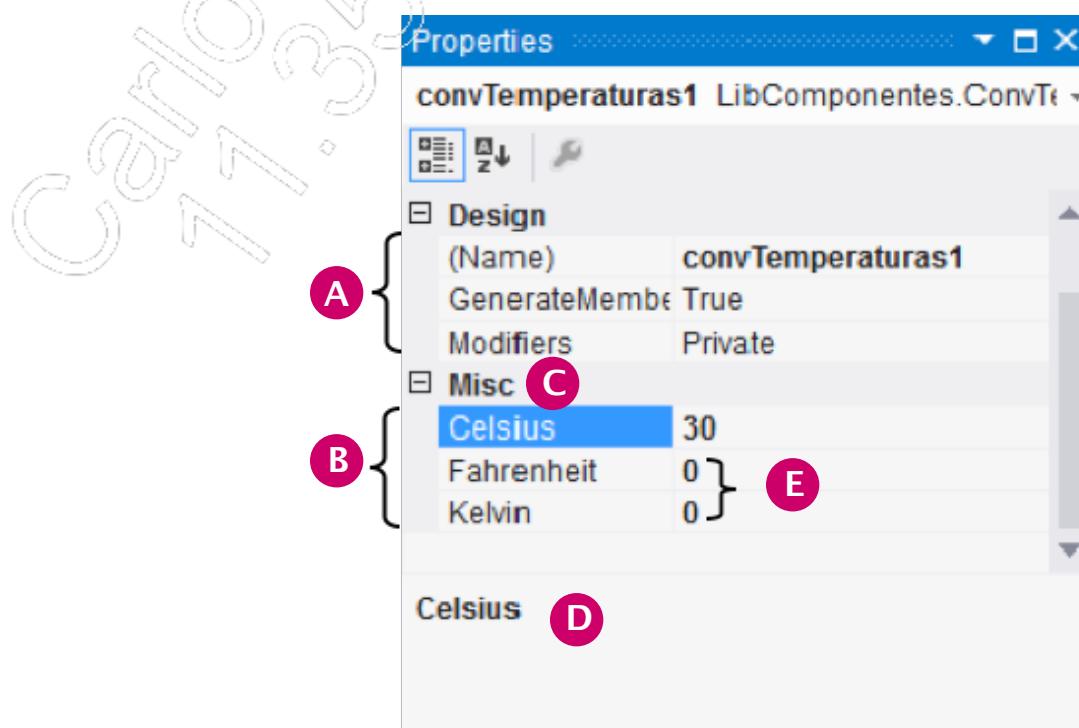


C# - Módulo I

7. Observe a ToolBox (A). Herdeiros diretos da classe **Component** são componentes não visuais, ou seja, não aparecem dentro do formulário (B), como **Timer**, **SaveFileDialog**, **ColorDialog**, etc;



8. Como herdeiros de **Component** aparecem na ToolBox e podem ser arrastados para o formulário, eles passam a interagir com a IDE do VS2012. Podemos, por exemplo, definir uma categoria, uma mensagem com explicação sobre a propriedade, etc. Para isso, usaremos atributos de propriedade:

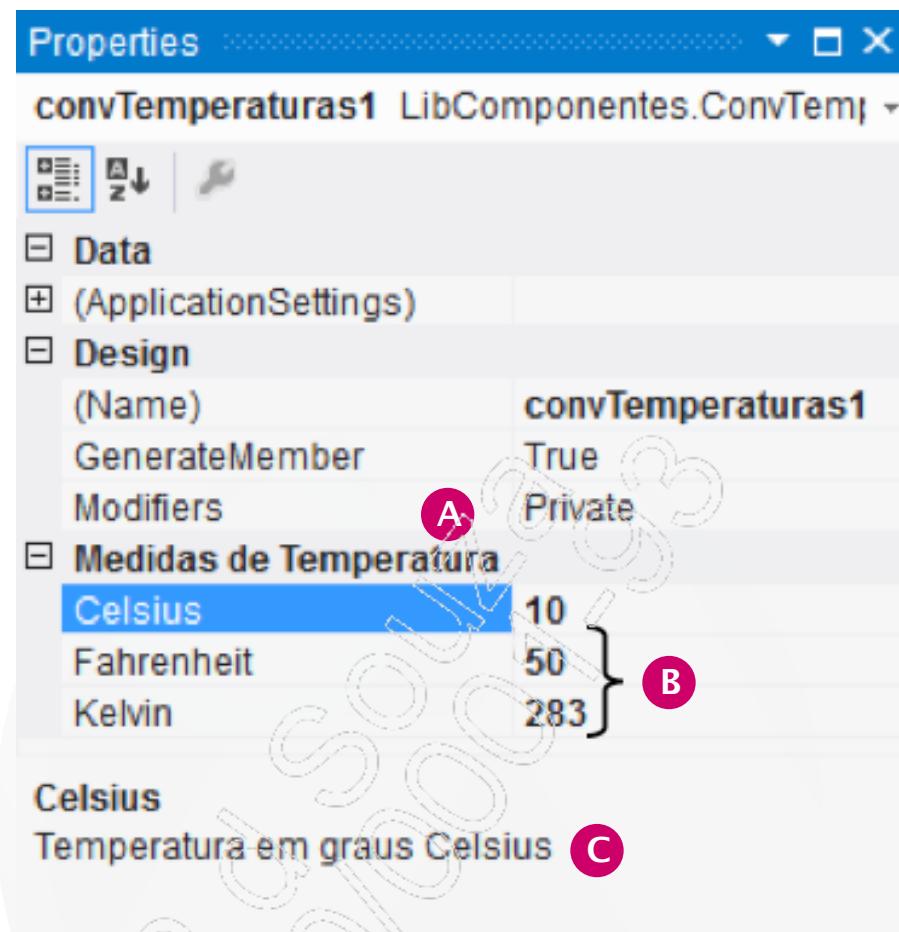


- A - Propriedades herdadas da classe **Component**;
- B - Propriedades adicionais criadas na classe herdeira;
- C - Categoria das propriedades. Este nome pode ser alterado;
- D - Descrição sobre a propriedade selecionada. Podemos definir uma descrição;
- E - A propriedade Celsius foi alterada, mas a janela de propriedades não foi atualizada para exibir os valores recalculados de Fahrenheit e Kelvin. Podemos também forçar esta atualização.

```
namespace LibBasico
{
    public class ConvTemperaturas : Component
    {
        // propriedades (CTRL + R + E para cada uma das variáveis)
        private decimal _celsius;

        //atributos de propriedade
        //definem características da propriedade em relação
        //a IDE do Visual Studio
        [Category ("Medidas de Temperatura"),
        Description ("Temperatura em Graus Celsius"),
        RefreshProperties(RefreshProperties.Repaint)]
        public decimal Celsius
        {
            get { return _celsius; }
            set
            {
                _celsius = value;
                _fahrenheit = (_celsius * 9 / 5) + 32;
                _kelvin = _celsius + 273;
            }
        }
    }
}
```

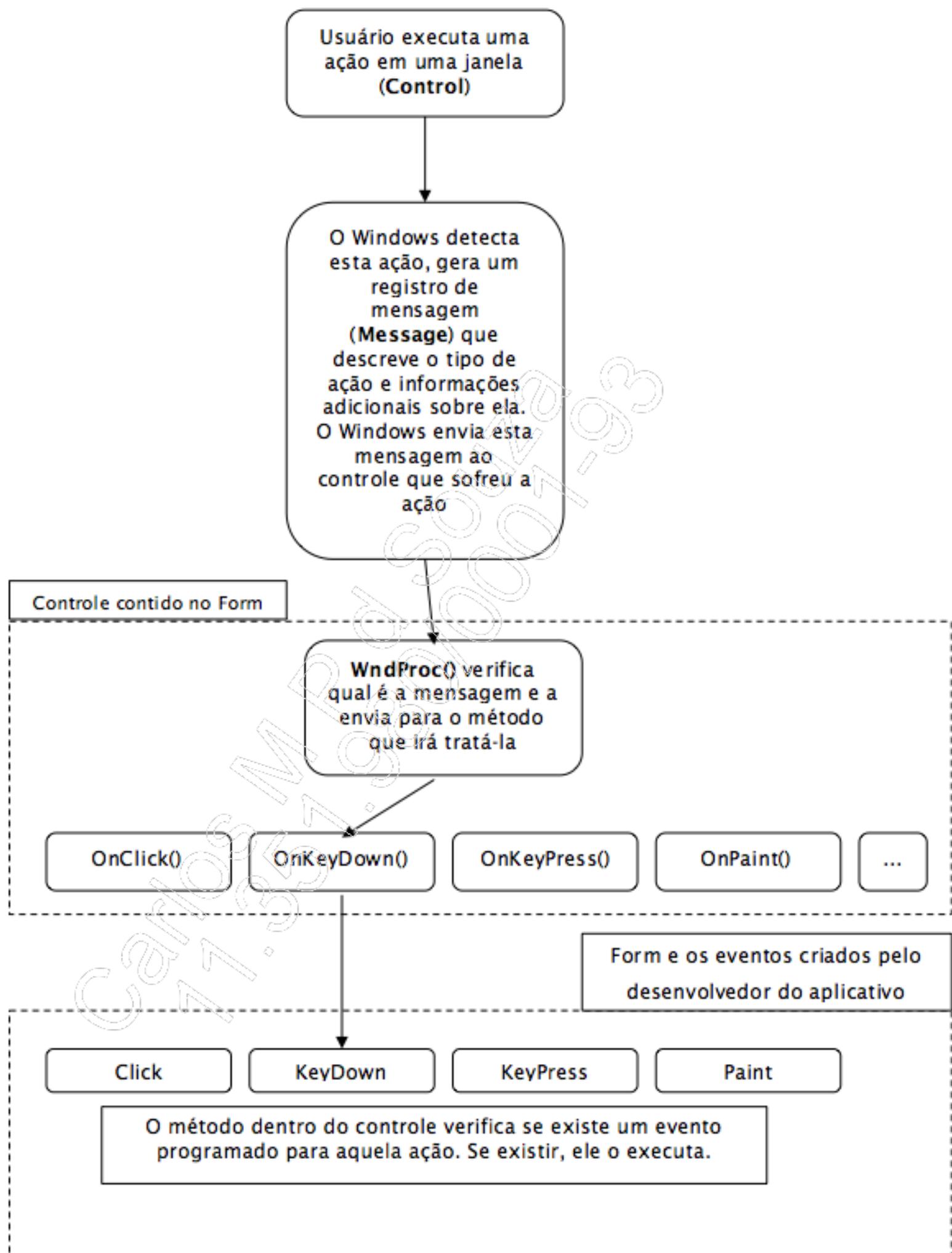
9. Coloque os mesmos atributos nas propriedades Fahrenheit e Kelvin, mudando apenas a descrição. Compile **LibComponentes** e volte ao projeto **TestaComponentes**:



- A - Novo nome de categoria;
- B - Propriedade Celsius foi alterada e as outras duas já apareceram imediatamente;
- C - Descrição sobre a propriedade Celsius.

9.7.7 Componentes visuais ou controles

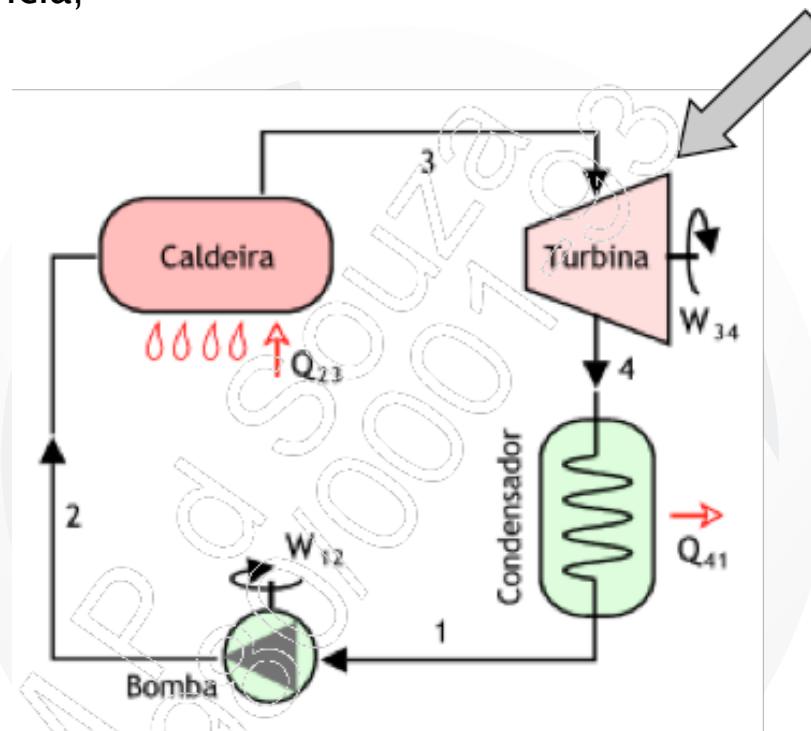
Todos os componentes visuais, como Button, TextBox, Label, entre outros, são herdeiros da classe **Control**. Como são visuais, as pessoas que os vêm na tela interagem com eles, digitando, clicando, arrastando ou qualquer outra das ações que fazemos diariamente quando utilizamos uma janela do Windows. Aliás, para o Windows, todos os componentes visuais são “janelas”. A plataforma .Net já incluiu na classe **Control** todo o mecanismo de detecção da ação exercida pelo usuário e o tratamento desta ação.



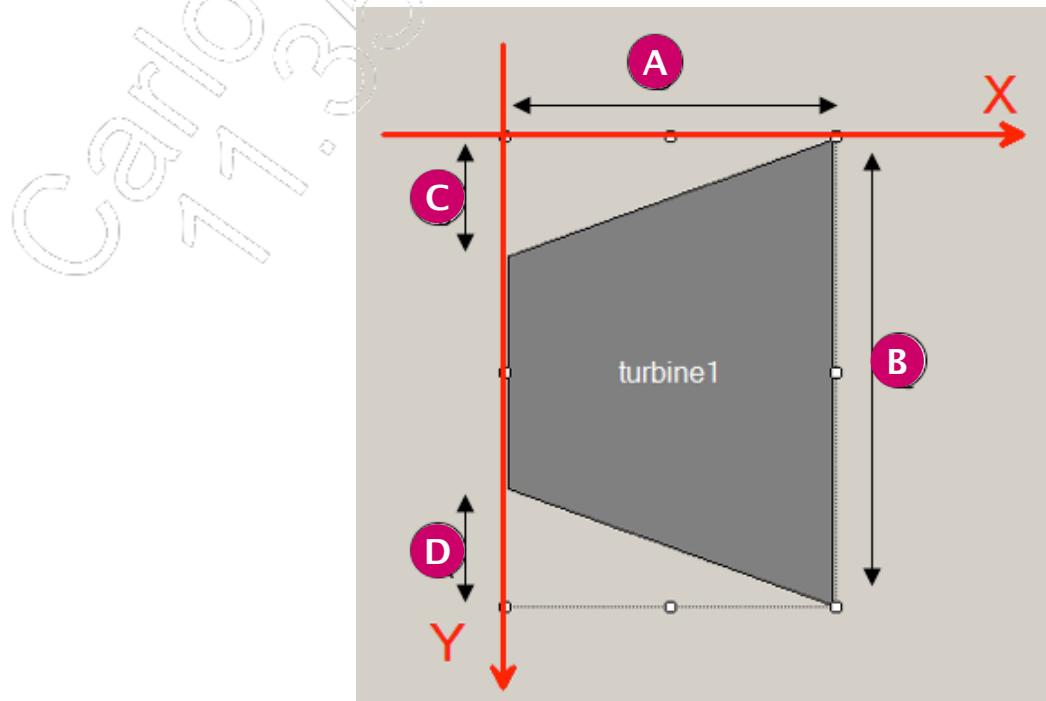
9.7.8. Exemplo: Herdeiros de Control

Neste exemplo, vamos aprender a herdar os componentes da classe **Control**:

1. Crie uma nova classe em **LibComponentes** e dê a ela o nome de **Turbina**. Em engenharia, existe um símbolo para representar uma turbina, mostrado na imagem a seguir. Imagine que usaremos o C# para montarmos uma tela que mostre a planta de uma usina com N turbinas. Desenhá-las uma a uma seria muito trabalhoso, visto que a turbina pode estar ligada ou desligada e isso muda a sua aparência;



- Imagem do componente Turbina:

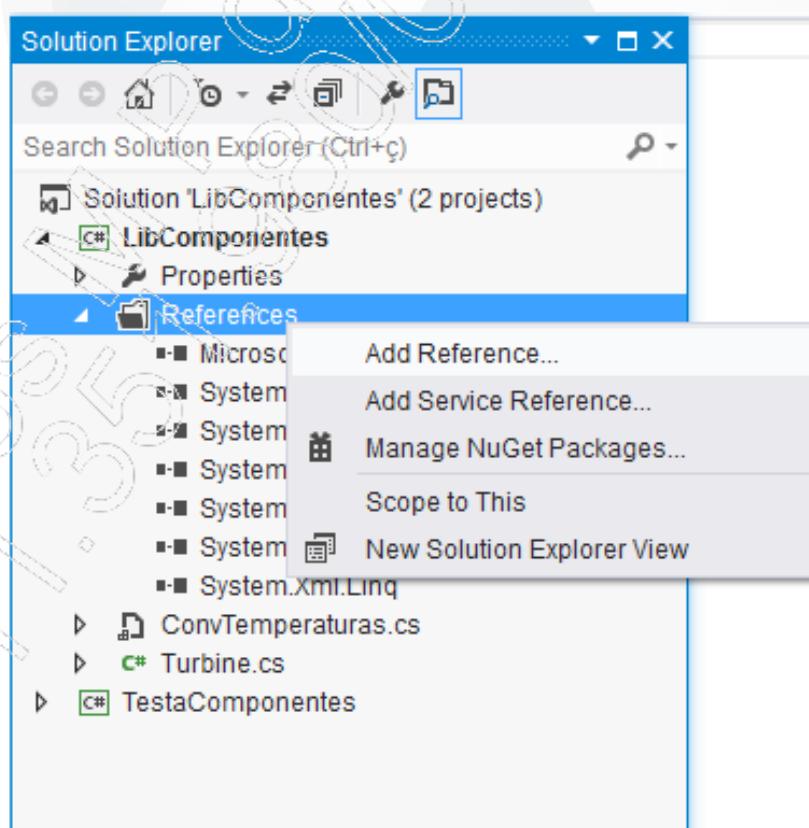


- A - Largura: Propriedade Width;
- B - Altura: Propriedade Height;
- C - Height / 4;
- D - Height / 4.

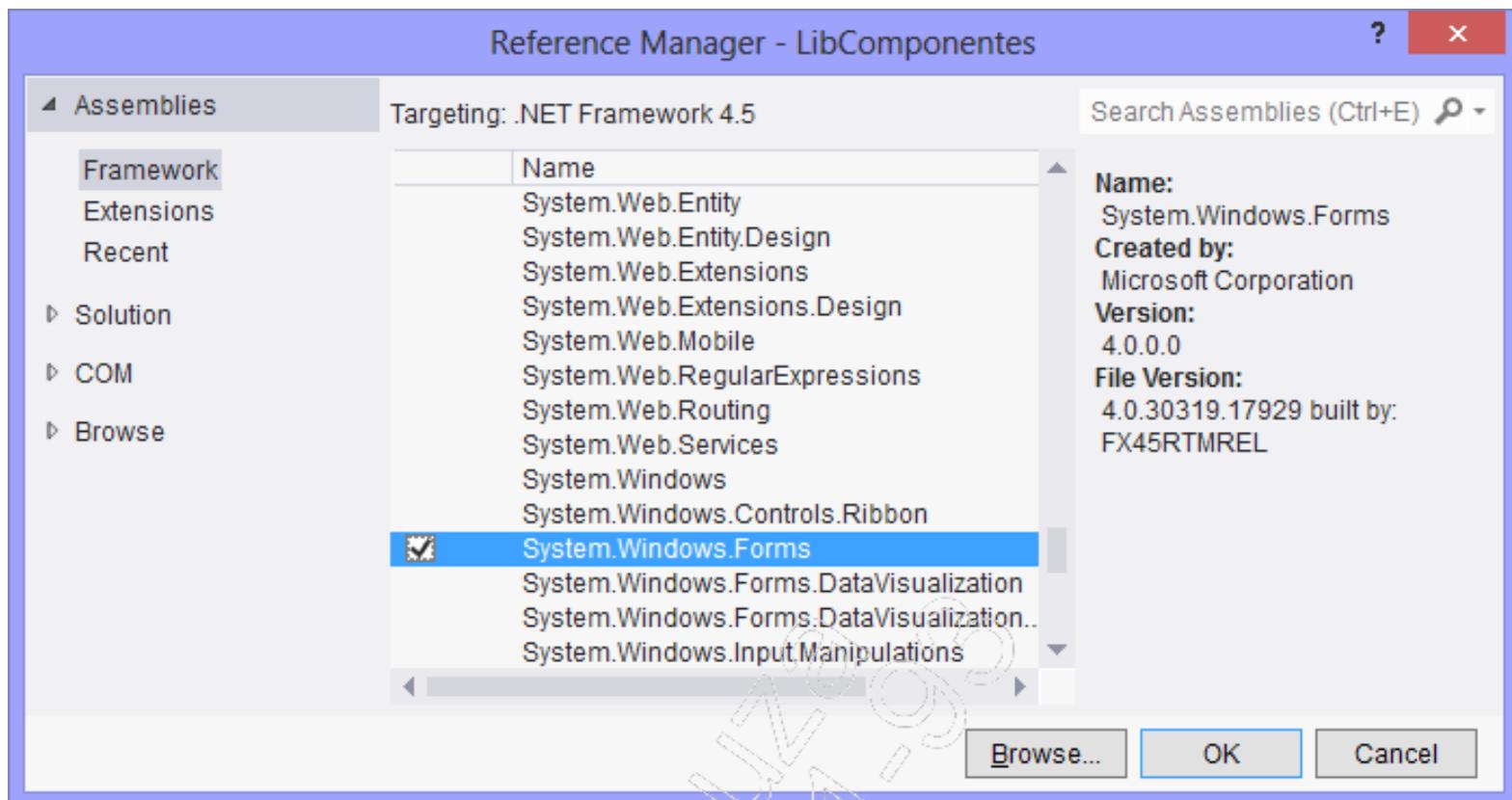
2. Adicione no item **References** de **LibComponentes** as seguintes bibliotecas (assemblies), disponíveis na plataforma .Net:

- **System.Windows.Forms**: Contém todo o suporte a componentes visuais (controles) para aplicações Windows Forms;
- **System.Drawing**: Contém as classes **Brush**, **Pen**, **Font**, **Color**, entre outras, que serão necessárias para fazermos o desenho.

3. Insira estes mesmos namespaces em **using**:



C# - Módulo I



The screenshot shows the 'Solution Explorer' and a code editor for the 'LibComponentes' project. In the Solution Explorer, the 'References' node is selected, showing dependencies on Microsoft.CSharp, System, System.Core, System.Data, System.Data.DataSetExtensions, System.Drawing, System.Windows.Forms, System.Xml, and System.Xml.Linq. The code editor displays the following C# code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using System.Drawing;
using System.Windows.Forms;

namespace LibComponentes
{
    class Turbine
    {
    }
}
```

4. Torne a classe **Turbine** pública e herdeira da classe **Control**;

```

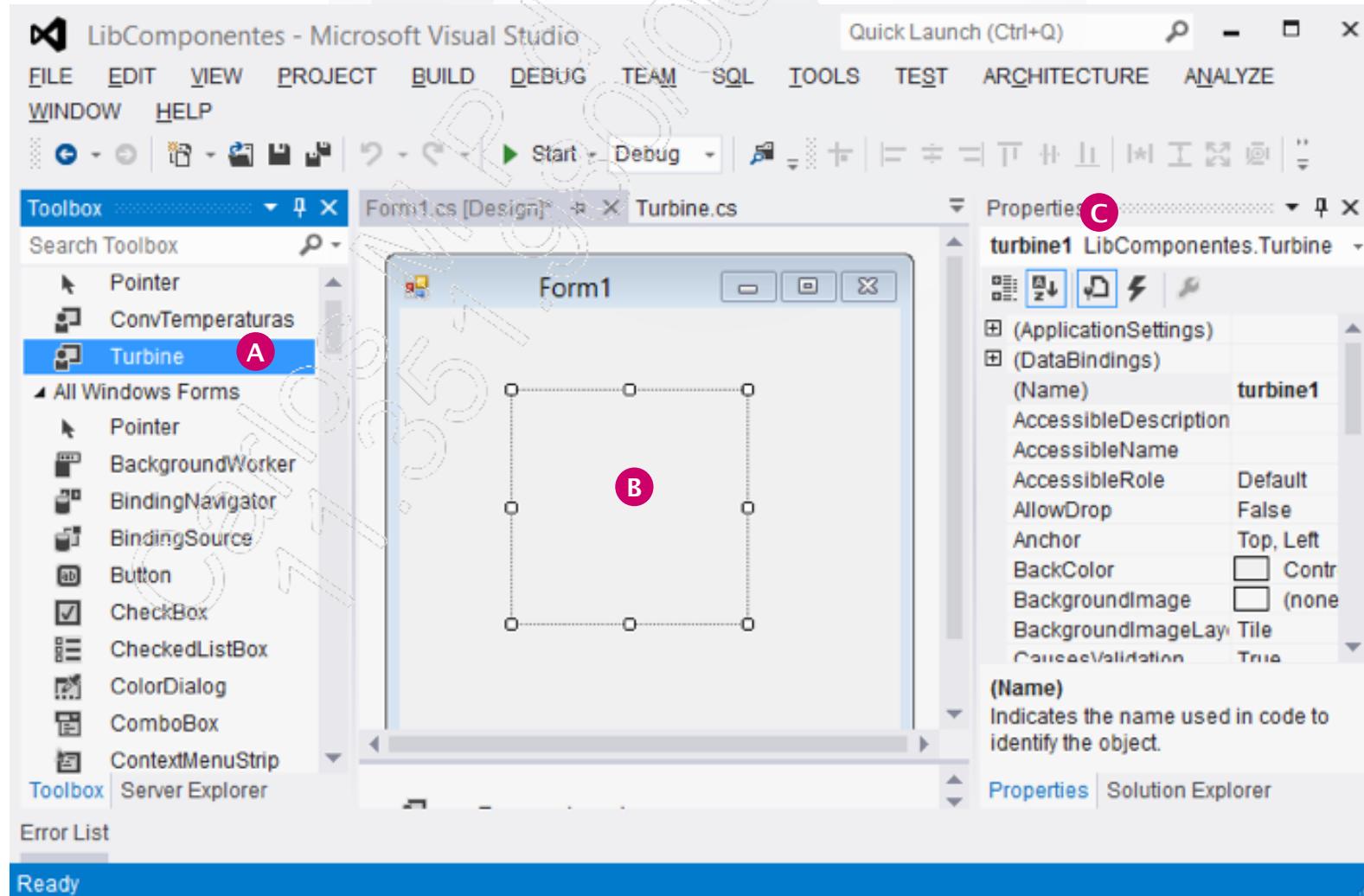
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using System.Drawing;
using System.Windows.Forms;

namespace LibComponentes
{
    public class Turbine : Control
    {
    }
}

```

5. Compile a biblioteca **LibComponentes** e depois selecione o formulário do projeto de teste. Observe que o componente **Turbine** já aparece na ToolBox e podemos trazê-lo para o formulário;



- A - Componente **Turbine**, herdeiro de **Control**, que, por sua vez, é herdeiro de **Component**, já aparece na ToolBox;
- B - Componente **Turbine** foi trazido para o formulário:
 - Como é herdeiro de **Control**, é visual e aparece na área do formulário;
 - Como é herdeiro direto de **Control**, não possui imagem predefinida, cabe a nós definirmos sua imagem;
 - O método **OnPaint()**, já existente na classe **Control**, é responsável por criar a atualizar a imagem do controle.
- C - Propriedade e eventos herdados da classe **Control**.

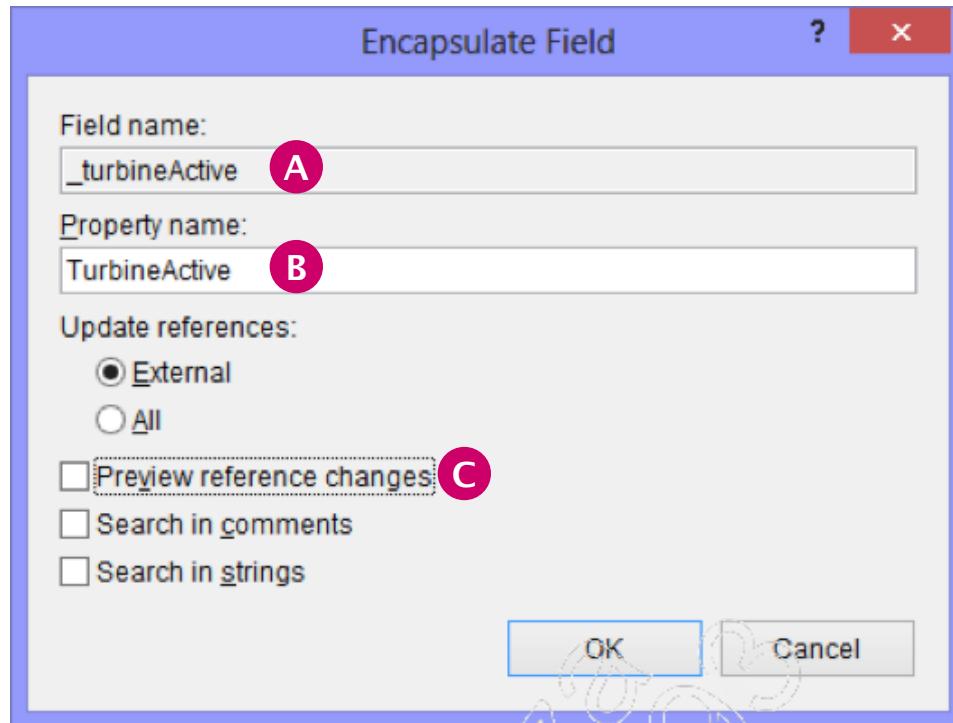
6. Crie as seguintes propriedades:

Nome	Tipo	Descrição
TurbineActive	bool	Indica se a turbina está ligada ou desligada.
TurbineColorOff	Color	Cor de fundo da turbina quando desligada.
TurbineColorOn	Color	Cor de fundo da turbina quando ligada.
TurbinePenColor	Color	Cor da linha de borda do desenho da turbina.
TurbinePenWidth	byte	Espessura da linha de borda.

! A partir da variável campo, pressione CTRL + R e CTRL + E para que o VS2012 declare a propriedade.

```
public class Turbine: Control
{
    /*
     * Variáveis campo para armazenar o conteúdo de cada propriedade
     * Com o cursor de texto posicionado na linha da variável,
     * pressione Ctrl + R, Ctrl + E e o Visual Studio declara
     * a propriedade a partir da variável campo
    */
    private bool _turbineActive;
    private Color _turbineColorOff;
    private Color _turbineColorOn;
    private Color _turbinePenColor;
    private byte _turbinePenWidth;
```

Cursor de texto da variável campo.
Pressionando CTRL + R e CTRL + E



- A - Nome da variável **campo**;
- B - Nome da propriedade que será criada;
- C - Apresenta um preview do que será feito, mas não é necessário.

7. Depois de tudo pronto, você pode comentar cada propriedade:

```
public class Turbine : Control
{
    /*
     * Variáveis campo para armazenar o conteúdo de cada propriedade
     * Com o cursor de texto posicionado na linha da variável,
     * pressione CTRL + R, CTRL + E e o Visual Studio declara
     * a propriedade a partir da variável campo
    */
    private bool _turbineActive;
    /// <summary>
    /// Indica se a turbina está ligada ou desligada
    /// </summary>
    public bool TurbineActive
    {
        get { return _turbineActive; }
        set { _turbineActive = value; }
    }
}
```

C# - Módulo I

```
private Color _turbineColorOff;
/// <summary>
/// Cor da turbina quando desligada
/// </summary>
public Color TurbineColorOff
{
    get { return _turbineColorOff; }
    set { _turbineColorOff = value; }
}

private Color _turbineColorOn;
/// <summary>
/// Cor da turbina quando ligada
/// </summary>
public Color TurbineColorOn
{
    get { return _turbineColorOn; }
    set { _turbineColorOn = value; }
}

private Color _turbinePenColor;
/// <summary>
/// Cor da linha de borda da turbina
/// </summary>
public Color TurbinePenColor
{
    get { return _turbinePenColor; }
    set { _turbinePenColor = value; }
}

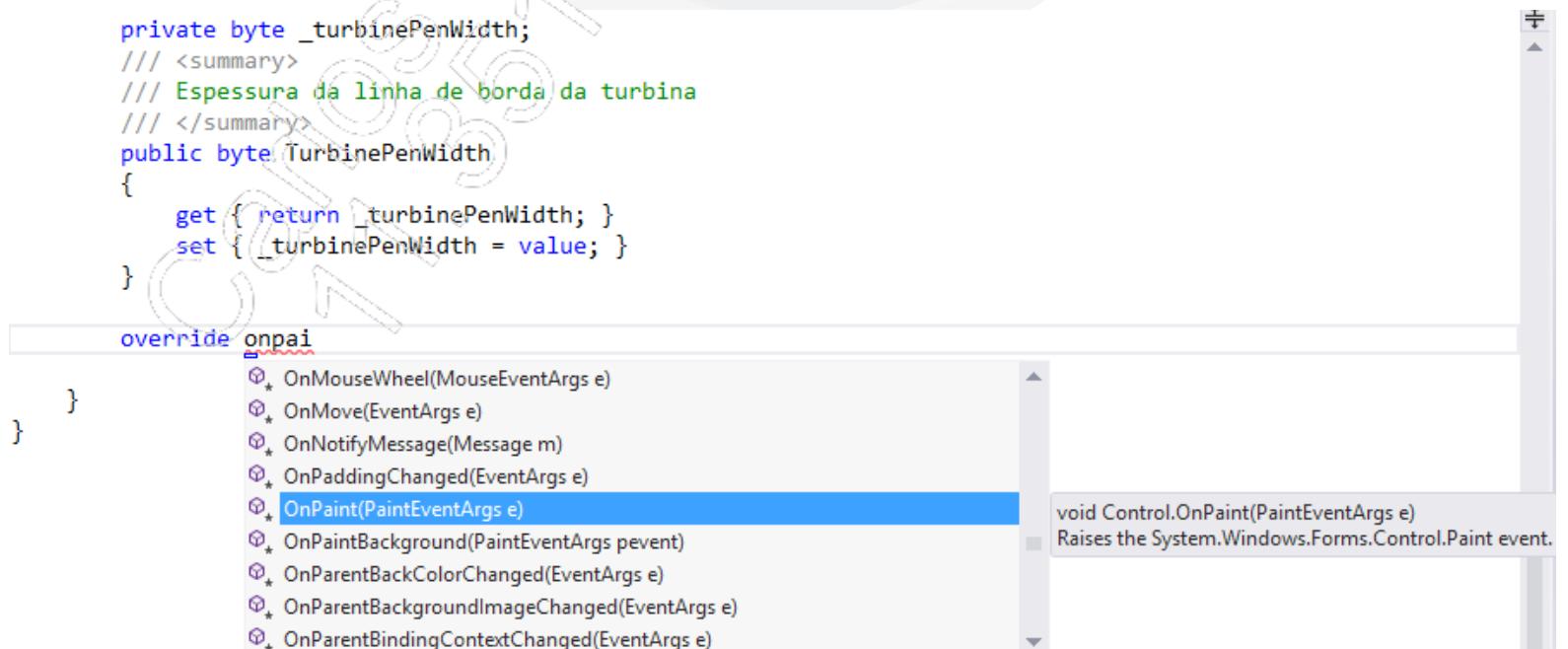
private byte _turbinePenWidth;
/// <summary>
/// Espessura da linha de borda da turbina
/// </summary>
public byte TurbinePenWidth
{
    get { return _turbinePenWidth; }
    set { _turbinePenWidth = value; }
}
```

8. Crie um construtor para definir um valor inicial para cada propriedade;

```
/// <summary>
/// Cria um objeto da classe Turbine
/// </summary>
public Turbine()
{
    _turbineActive = false;
    _turbineColorOff = Color.Gray;
    _turbineColorOn = Color.Blue;
    _turbinePenColor = Color.Black;
    _turbinePenWidth = 1;

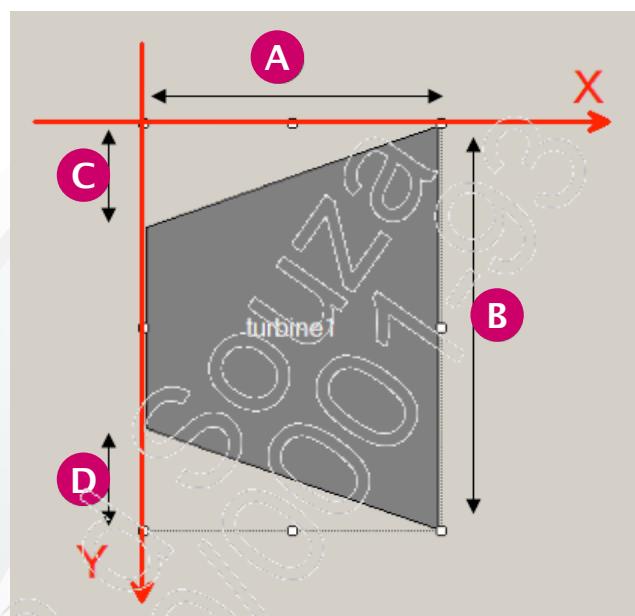
    Width = 70;
    Height = 120;
}
```

9. Como já comentado anteriormente, o método **OnPaint()** da classe **Control** é executado automaticamente pelo Windows (que envia uma mensagem **WM_PAINT** para o controle). É esse método que desenha o conteúdo do controle. Então, para que a turbina seja desenhada, precisamos criar um método **OnPaint()** que complemente o **OnPaint()** herdado da classe **Control**, uma operação conhecida como **override**. Na classe **Turbine**, digite a palavra **override** e dê um espaço:



```
protected override void OnPaint(PaintEventArgs e)
{
    // executa o método OnPaint da classe Control
    base.OnPaint(e);
    // código complementar que queremos executar
}
```

10. Complemente o método desenhando a turbina;



```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    // cor de fundo da turbina dependendo se estiver ligada ou desligada
    Brush br = new SolidBrush(
        _turbineActive ? _turbineColorOn : _turbineColorOff);

    // linha de borda
    Pen linha = new Pen(_turbinePenColor, _turbinePenWidth);
    // pontos que formarão o polígono (trapézio)
    Point[] pts = { new Point(0,Height/4),
                    new Point(Width-1, 0),
                    new Point(Width-1, Height),
                    new Point(0, 3*Height/4) };

    // desenhar o preenchimento do polígono
    e.Graphics.FillPolygon(br, pts);
    // desenhar a borda do polígono
    e.Graphics.DrawPolygon(linha, pts);
```

```
    e.Graphics.DrawPolygon(linha, pts);
    // Exibir o conteúdo da propriedade Text centralizado
    // medir o tamanho do texto para poder mostrar centralizado
    SizeF tam = e.Graphics.MeasureString(Text, Font);
    // mostrar o texto centralizado
    e.Graphics.DrawString(Text, Font,
        new SolidBrush(ForeColor), // cor da letra
        (Width - tam.Width) / 2, // posição X
        (Height - tam.Height) / 2); // posição Y
}
```

11. Compile **LibComponentes** e retorne ao formulário do projeto de teste. Altere as propriedades para ver o seu efeito. Note que, no momento em que alteramos uma das propriedades na janela de propriedades, a imagem da turbina não se altera, a não ser que a gente ponha o ponteiro do mouse sobre ela e arraste. É assim que funciona o **OnPaint()**, ele só é executado quando o controle sofre uma ação direta do usuário. Para forçar a execução do **OnPaint()** via programação, a classe **Control** possui um método chamado **Invalidate()**. Altere os métodos SET de todas as propriedades incluindo o **Invalidate**:

```
/// <summary>
/// Cor de fundo quando a turbina estiver desligada
/// </summary>
public Color TurbineColorOff
{
    get { return _turbineColorOff; }
    set
    {
        _turbineColorOff = value;
        // força a execução do OnPaint quando a propriedade
        // for alterada
        Invalidate();
    }
}
```

12. Realize o mesmo procedimento indicado no passo anterior para as outras propriedades;

13. Depois disso, você vai perceber que o mesmo efeito ocorre com a propriedade **Text**, ou seja, quando o conteúdo desta propriedade é alterado na janela de propriedades, ele não é exibido até que o controle seja arrastado. A solução será a mesma, mas onde está o método SET da propriedade **Text**? Ele também está codificado na classe **Control**. Na classe **Turbina**, precisaremos fazer override da propriedade **Text**:

The screenshot shows a code editor with the following code:

```
public override string Text
{
    get
    {
        return base.Text;
    }
    set
    {
        base.Text = value;
        // incluir o Invalidate()
        Invalidate();
    }
}
```

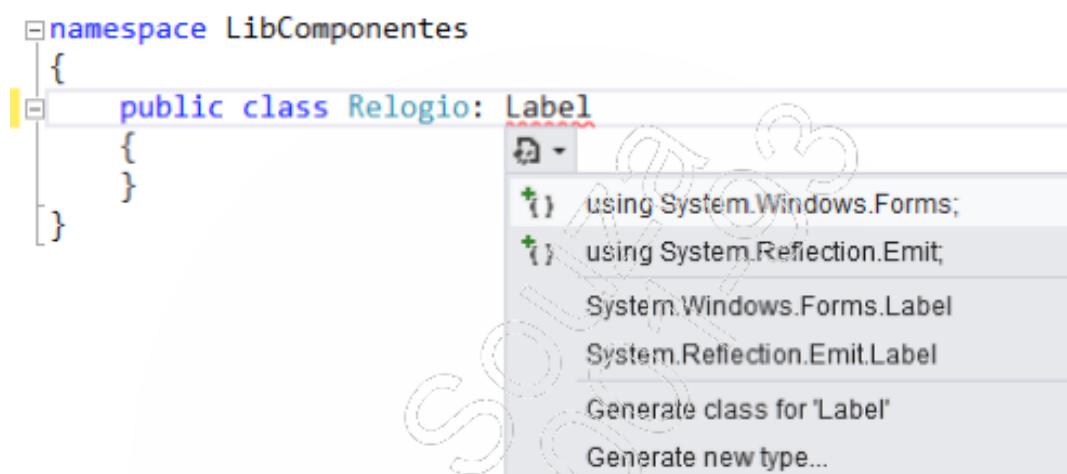
A tooltip is displayed above the code, listing various methods of the **Control** class. The **Text** method is highlighted with a blue background, indicating it is the current target for completion or modification.

Lembrando que fazer override significa recriar um método ou propriedade herdada, adicionando a ela mais funcionalidade, sem perder as funções que ela já executa na classe ancestral.

9.7.9. Exemplo: Herdeiros de controles já existentes

Neste exemplo, criaremos um herdeiro da classe **Label** que mostrará data e hora atuais:

1. Crie uma nova classe chamada **Relogio** em **LibComponentes**. Torne-a pública e herdeira de **Label**:



2. Precisaremos de um timer para atualizar o texto do label a cada segundo. Também criaremos um tipo enumerado para determinar o formato de apresentação da data e hora:

```
public class Relogio: Label
{
    // declaração do objeto timer
    private Timer timer;

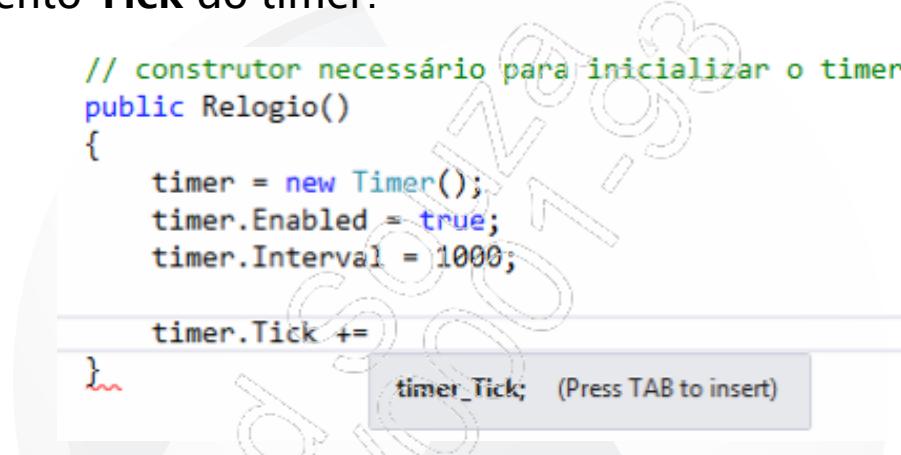
    // tipo de dado “enumerado” para definir as opções possíveis
    // de formatação do relógio
    // Um tipo enumerado serve para declarar variáveis
    // que vão aceitar somente um dos elementos do tipo
    public enum FormatoRelogio
    {
        HoraCurto,           // 09:37:38
        DataHoraCurto,       // 23/03/2011 - 09:37:38
        DataHoraComDSAbrev, // qua - 23/03/2011 - 09:37:38
        DataHoraComDSExtenso, // quarta-feira - 23/03/2011 - 09:37:38
        Customizado          // definido pelo usuário
    }
}
```

3. Defina variáveis campo para as propriedades que vão definir o formato de apresentação da data e hora;

```
// propriedade para definir o formato de apresentação da hora/data  
private FormatoRelogio _formato;  
// propriedade para definir o formato customizado da data/hora  
private string _formatoCustomizado;
```

4. Utilize CTRL + R e CTRL + E para criar as propriedades;

5. Utilize o método construtor para inicializar o timer. Pressione TAB + TAB para completar o evento Tick do timer:



```
// construtor necessário para inicializar o timer  
public Relogio()  
{  
    timer = new Timer();  
    timer.Enabled = true;  
    timer.Interval = 1000;  
  
    timer.Tick +=
```

- Resultado:

```
// construtor necessário para inicializar o timer  
public Relogio()  
{  
    timer = new Timer();  
    timer.Enabled = true;  
    timer.Interval = 1000;  
  
    timer.Tick += timer_Tick;
```



```
void timer_Tick(object sender, EventArgs e)  
{  
    throw new NotImplementedException();  
}
```

6. Complete o método **timer_Tick**:

```
void timer_Tick(object sender, EventArgs e)
{
    string mascara = _formatoCustomizado;

    switch (_formato)
    {
        case FormatoRelogio.HoraCurto:
            mascara = "hh:mm:ss";
            break;
        case FormatoRelogio.DataHoraCurto:
            mascara = "dd/MM/yyyy - hh:mm:ss";
            break;
        case FormatoRelogio.DataHoraComDSAbrev:
            mascara = "ddd - dd/MM/yyyy - hh:mm:ss";
            break;
        case FormatoRelogio.DataHoraComDSExtenso:
            mascara = "dddd - dd/MM/yyyy - hh:mm:ss";
            break;
    }
    Text = DateTime.Now.ToString(mascara);
}
```

7. Suponha que a pessoa que usa o componente **Relogio** queira executar alguma ação adicional toda vez que o timer do relógio executar a sua ação. Na verdade, você quer permitir que o **Relogio** seja usado também como **Timer**. Para isso, precisa criar um evento **Tick** (poderia ser outro nome) para o **Relogio**:

```
public class Relogio: Label
{
    // declaração do objeto timer
    private Timer timer;

    // criar o evento Tick. Usaremos o delegate EventHandler
    // que é o mesmo que define o Tick do Timer
    public event EventHandler Tick;
```

C# - Módulo I

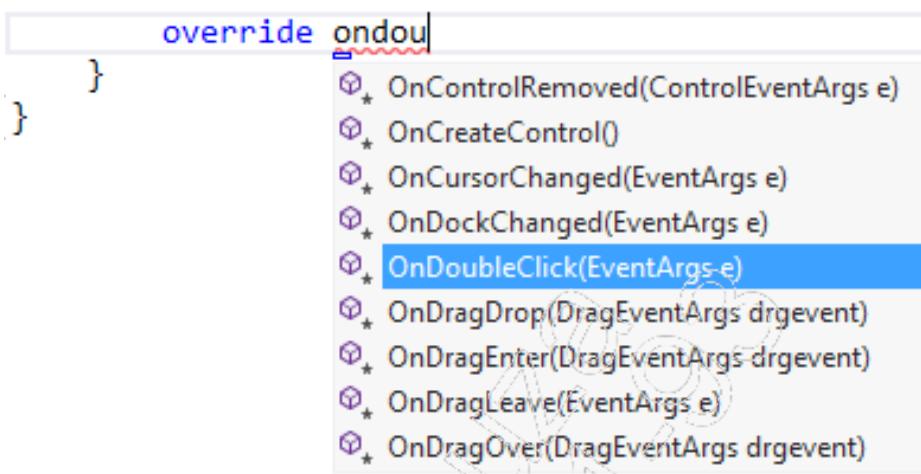
8. Caso o evento aponte para um método, este método deverá ser executado sempre que o relógio for atualizado. Você deve incluir isso no método **timer_Tick**:

```
void timer_Tick(object sender, EventArgs e)
{
    string mascara = _formatoCustomizado;
    switch (_formato)
    {
        case FormatoRelogio.HoraCurto:
            mascara = "hh:mm:ss";
            break;
        case FormatoRelogio.DataHoraCurto:
            mascara = "dd/MM/yyyy - hh:mm:ss";
            break;
        case FormatoRelogio.DataHoraComDSAbrev:
            mascara = "ddd - dd/MM/yyyy - hh:mm:ss";
            break;
        case FormatoRelogio.DataHoraComDSExtenso:
            mascara = "dddd - dd/MM/yyyy - hh:mm:ss";
            break;
    }
    Text = DateTime.Now.ToString(mascara);
    if (Tick != null)
        Tick(this, e);
}
```

9. Como já mostramos anteriormente, todo evento de um controle possui um método interno da classe responsável pela sua execução:

Evento	Método Interno
Click	OnClick
DoubleClick	OnDoubleClick
KeyDown	OnKeyDown
KeyPress	OnKeyPress
Paint	OnPaint
Tick	OnTick
...	...

10. Quando você cria um herdeiro de uma classe que possui eventos e quer que a classe herdeira já execute alguma ação padrão para um dado evento, o herdeiro não deve programar o evento, mas sim fazer override do método responsável pelo evento. Suponha que queira parar ou reiniciar o relógio quando ele sofrer um duplo-clique. O correto será fazer override do método **OnDoubleClick** da classe **Label**:



```

    override void ondou
    {
        base.OnControlRemoved(e);
        // código adicional
        timer.Enabled = !timer.Enabled;
    }
}

protected override void OnDoubleClick(EventArgs e)
{
    base.OnDoubleClick(e);
    // código adicional
    timer.Enabled = !timer.Enabled;
}

```

11. Seguindo o raciocínio anterior, se alguém criar um herdeiro da classe **Relogio**, não poderá fazer algo semelhante com o evento **Tick** do **Relogio**:

- O método **timer_Tick** é private, só é visível dentro da classe **Relogio**;
- Para poder sofrer um override, o método precisa ser virtual;
- Solução para o problema:

```

void timer_Tick(object sender, EventArgs e)
{
    string mascara = _formatoCustomizado;
    switch (_formato)
    {
        case FormatoRelogio.HoraCurto:

```

C# - Módulo I

```
mascara = "hh:mm:ss";
break;
case FormatoRelogio.DataHoraCurto:
    mascara = "dd/MM/yyyy - hh:mm:ss";
    break;
case FormatoRelogio.DataHoraComDSAbrev:
    mascara = "ddd - dd/MM/yyyy - hh:mm:ss";
    break;
case FormatoRelogio.DataHoraComDSExtenso:
    mascara = "dddd - dd/MM/yyyy - hh:mm:ss";
    break;
}
Text = DateTime.Now.ToString(mascara);
// executa o método que testa e executa o evento Tick
OnTick(e);
}
// método para testar e executar o evento Tick
// protected: visível nos herdeiros, mas não pela instância da classe
// virtual: permite override
protected virtual void OnTick(EventArgs e)
{
    if (Tick != null) Tick(this, e);
}
```

12. Para finalizar este controle, note que, para quem usa a classe **Relogio**, a propriedade **Text** não tem utilidade alguma. Se digitarmos algo na propriedade **Text** em no máximo 1 segundo, ele será substituído pela hora atual. Você pode então ocultar a propriedade **Text** da janela de propriedades. Faça override da propriedade **Text** e inclua o atributo **Browsable**.

```
// using System.ComponentModel
[Browsable(false)]
public override string Text
{
    get
    {
        return base.Text;
    }
    set
    {
        base.Text = value;
    }
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Uma classe, assim como uma estrutura, define as características dos objetos pertencentes a ela. Essas características ou atributos são suas propriedades (dados que ela manipula) e seus métodos (ações que ela executa);
- A herança possibilita que as classes compartilhem seus atributos, métodos e outros membros de classe entre si. Para a ligação entre as classes, a herança adota um relacionamento esquematizado hierarquicamente;
- Ao utilizarmos um método de extensão, poderemos ampliar uma classe ou uma estrutura já existente com métodos estáticos adicionais. Dessa forma, os métodos estáticos podem ser utilizados ao longo do código, seja qual for a instrução que faça referência aos dados do tipo estendido;
- Um construtor é um método executado automaticamente toda vez que instanciamos um objeto, ou seja, ele é responsável por inicializar os membros de dados do objeto que está sendo criado;
- Os métodos são rotinas que realizam tarefas. Eles podem ou não retornar valores e receber parâmetros;
- Polimorfismo é um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de forma específica para cada uma das classes derivadas.

9

Classes e estruturas

Teste seus conhecimentos

Carlos M. 77.357.900-007-303 Souza



IMPACTA
EDITORA

1. Assinale a alternativa que completa adequadamente a frase a seguir:

O método _____ é responsável pela criação e inicialização de um objeto de uma determinada classe.

- a) override
- b) Create
- c) Init
- d) construtor
- e) overload

2. Assinale a alternativa que completa adequadamente a frase a seguir:

Para que possamos fazer _____ de um método, é necessário que ele seja _____.

- a) override, dynamic
- b) overload, dynamic
- c) overload, virtual
- d) override, virtual
- e) herança, virtual

3. Assinale a alternativa que completa adequadamente a frase a seguir:

O método _____ de uma propriedade é responsável por retornar o seu valor, já o método _____ atribui valor a ela.

- a) construtor, destrutor
- b) destrutor, construtor
- c) get, set
- d) return, set
- e) return, get

4. Qual das alternativas a seguir está correta sobre sobrecarga de método (overload)?

- a) Complementa as ações de um método herdado de uma classe ancestral.
- b) Podemos ter dois ou mais métodos com o mesmo nome, mas com parâmetros diferentes.
- c) Podemos ter dois ou mais métodos com o mesmo nome, mas com parâmetros e tipo de retorno diferentes.
- d) Em classes derivadas, podemos ter métodos com o mesmo nome e a mesma assinatura, mas que executam tarefas distintas.
- e) O C# não possui este recurso.

5. Qual das alternativas a seguir está correta sobre polimorfismo?

- a) Complementa as ações de um método herdado de uma classe ancestral.
- b) Podemos ter dois ou mais métodos com o mesmo nome, mas com parâmetros diferentes.
- c) Podemos ter dois ou mais métodos com o mesmo nome, mas com parâmetros e tipo de retorno diferentes.
- d) Em classes derivadas, podemos ter métodos com o mesmo nome e a mesma assinatura, mas que executam tarefas distintas.
- e) O C# não possui este recurso.

9

Classes e estruturas

Mãos à obra!

Carlos M.
77.357.9001
SOUZA
07-993



IMPACTA
EDITORA

Laboratório 1

A - Criando uma classe herdeira

1. Crie na biblioteca **LibComponentes** uma classe chamada **BlinkLabel**, herdeira da classe **Label**:

- **Finalidade:** Alternar a cor da letra do texto exibido pelo **BlinkLabel** em intervalos regulares de tempo;
- **Propriedades:**
 - **Active** do tipo **bool**: Se true, o label estará piscando; caso contrário, não;
 - **ForeColorOn**: do tipo **Color**: Cor da letra para alternar;
 - **ForeColorOff**: do tipo **Color**: Cor da letra para alternar;
 - **Interval** do tipo **int**: Intervalo de tempo (em milissegundos) para alternância da cor da letra.
- **Construtor:**
 - Deve criar a instância do Timer que vai controlar a alternância das cores;
 - Dar valor inicial às propriedades;
 - Definir o evento **Tick** do timer.
- **Evento Tick do timer:**

```
if (ForeColor == _foreColorOff)
    ForeColor = _foreColorOn;
else
    ForeColor = _foreColorOff;
```

2. Oculte a visibilidade da propriedade **ForeColor** na janela de propriedades;

3. Crie o evento **Tick** para o **BlinkLabel** de modo que seja possível executar uma outra ação sincronizada com a alternância das cores. Deve permitir que herdeiros de **BlinkLabel** possam fazer override do método responsável pelo evento.

Expressões regulares

10

- ✓ Principais símbolos usados em uma expressão regular;
- ✓ A classe Regex;
- ✓ Exemplo: Projeto para testar expressões regulares.



IMPACTA
EDITORA

10.1. Introdução

Uma expressão regular é uma string formada por símbolos especiais que nos permite validar se um dado está no formato correto, por exemplo:

- A placa de um automóvel deve começar com três letras maiúsculas seguidas de um traço e, em seguida, quatro números;
- CEP é formado por quatro números, um traço e mais três números;
- O CPF deve obedecer ao formato 999.999.999-99.

10.2. Principais símbolos usados em uma expressão regular

Podemos utilizar diversos símbolos em uma expressão regular. Vamos conhecer os mais importantes:

SÍMBOLO	FINALIDADE
.	<p>(ponto)</p> <p>Equivale a um único caractere qualquer. Expressão: M.LA Strings válidas: MALA, MELA, MULA, MOLA. Strings inválidas: MOELA, MIELA, mala, mula, mola.</p>
\$	<p>Testa se a string termina com a palavra que antecede o símbolo: Expressão: legal\$ String válida: o curso foi legal String inválida: o curso foi legal?</p>
^	<p>Testa se a string começa com a palavra que sucede ao símbolo. Expressão: ^onde String válida: onde está o livro String inválida: não sei onde está</p>
*	<p>Qualquer quantidade (inclusive zero) do caractere que precede o símbolo. Expressão: A0*B Strings válidas: AB, A0B, A00B, A000B Strings inválidas: BA, ABC, A001B, A000</p>

SÍMBOLO	FINALIDADE
+	Uma ou mais repetições do caractere que precede o símbolo. Expressão: A0+B Strings válidas: A0B, A00B, A000B Strings inválidas: AB, BA, ABC, A001B, A000
?	Zero ou uma ocorrência do caractere que precede o símbolo. Expressão: A0?B Strings válidas: AB, A0B, ABC Strings inválidas: BA, A001B, A000
[] [c1-c2] [^c1-c2]	Lista de opções para o caractere naquela posição. Expressão: m[iou]a Strings válidas: mia, moa, mua Strings inválidas: mea, moua Faixa de caracteres válidos (de “a” até “z”). Expressão: [a-z]* Strings válidas: magno, impacta, abelha Strings inválidas: anão, porta-retratos, d'avila Faixa de caracteres inválidos (não contidos na faixa de “a” até “z”). Expressão: [^a-z]* Strings válidas: 123-@!", ABC123 Strings inválidas: a123, 123g
	Ou [A B][0-9]: Letra A ou B seguida de um número de 0 até 9. [A B][0-9]+: Letra A ou B seguida de um ou mais números de 0 até 9. [A B][0-9]*: Letra A ou B seguida ou não de números de 0 até 9.
0	Conjunto Pares de letra e número repetidos uma ou mais vezes: ([A-Z][0-9]) Número hexadecimal, uma ou mais repetições das letras de A a F ou dos números de 0 a 9: ([A-F][0-9])+
{n} {de, até}	Quantidade de repetições do caractere ou conjunto que precede o símbolo. Sequência de três números: [0-9]{3} Sequência de 2 a 4 números: [0-9]{2,4}

10.3. A classe Regex

Esta classe é utilizada para validar uma string com base em uma expressão regular. Ela é composta de vários métodos para fazer diversos tipos de validação.

- **Construtores:**

- **Regex():** Cria um objeto da classe **Regex**;
- **Regex(string expressao):** Cria um objeto da classe **Regex** usando a expressão regular especificada;
- **Regex(string expressão, RegexOptions opções):** Cria um objeto da classe **Regex** usando a expressão regular e as opções especificadas.

Vejamos o exemplo a seguir:

```
// define sem nenhuma opção de teste
RegexOptions opt = RegexOptions.None;
// se o checkBox estiver selecionado, define não diferenciar
// minúsculas de maiúsculas
if (ckbIgnoreCase.Checked) opt = RegexOptions.IgnoreCase;
// cria instância de Regex
Regex regex = new Regex("[A-Z]{3}-[0-9]{4}", opt);
```

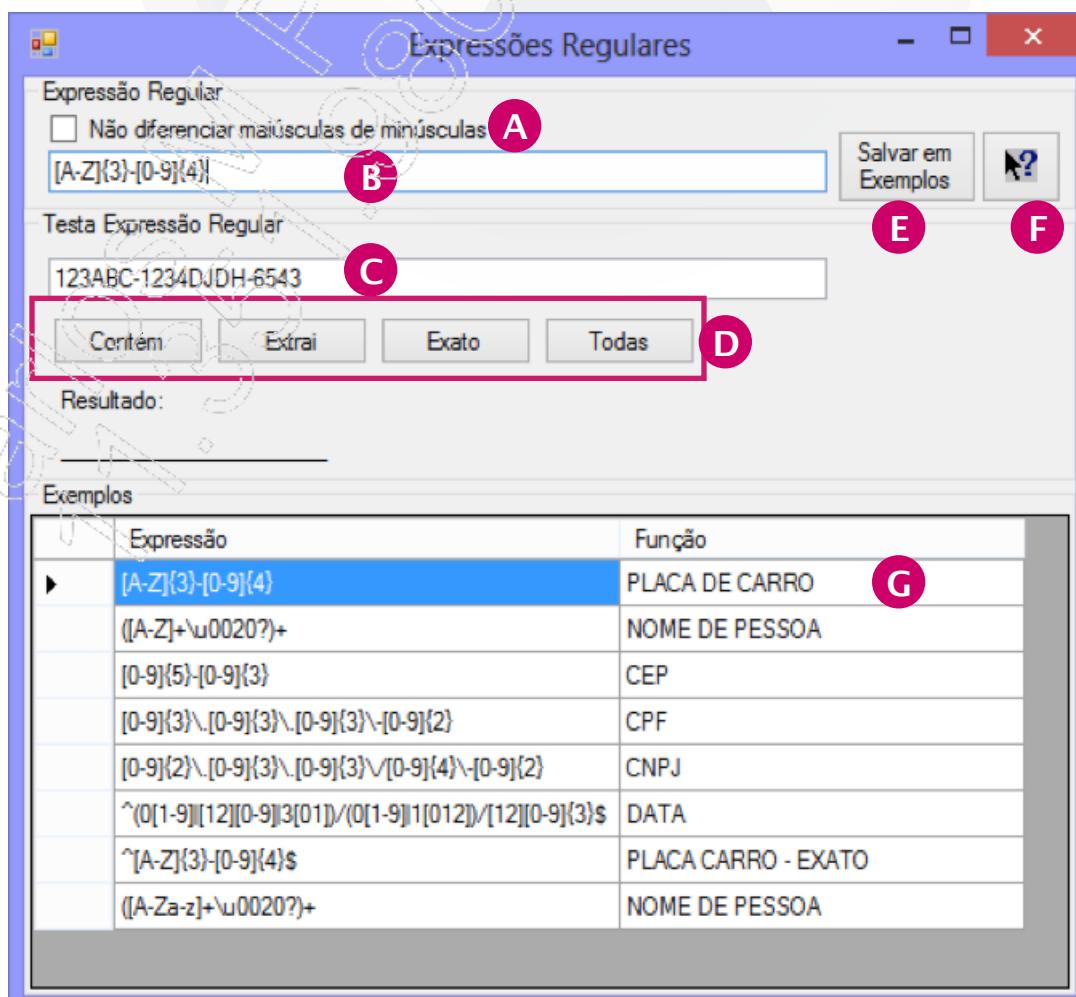
- **Métodos:**

- **bool IsMatch(string texto):** Indica se o texto tem correspondência com a expressão regular definida no construtor;
- **bool IsMatch(string texto, string exprReg):** Indica se o texto tem correspondência com a expressão regular definida em **exprReg**;
- **bool IsMatch(string texto, string exprReg, RegexOptions opt):** Indica se o texto tem correspondência com a expressão regular definida em **exprReg** de acordo com as opções definidas em **opt**;
- **Match Match(string texto):** Indica se o texto tem correspondência com a expressão regular definida no construtor;
- **Match Match(string texto, string exprReg):** Indica se o texto tem correspondência com a expressão regular definida em **exprReg**;

- **Match Match(string texto, string exprReg, RegexOptions opt)**: Indica se o texto tem correspondência com a expressão regular definida em **exprReg** de acordo com as opções definidas em **opt**;
- **MatchCollection Matches(string texto)**: Retorna com todas as correspondências que o texto tem com a expressão regular definida no construtor;
- **MatchesCollection Matches(string texto, string exprReg)**: Retorna com todas as correspondências que o texto tem com a expressão regular definida em **exprReg**;
- **MatchCollection Matches(string texto, string exprReg, RegexOptions opt)**: Retorna com todas as correspondências que o texto tem com a expressão regular definida em **exprReg** de acordo com as opções definidas em **opt**.

10.4. Exemplo: Projeto para testar expressões regulares

Vejamos o seguinte exemplo de teste de expressões regulares:



- **A** - Não diferencia minúsculas de maiúsculas no teste;
- **B** - Expressão regular que define o formato do texto;
- **C** - Texto que será testado pela expressão regular;
- **D** - Botões que farão os testes:
 - **Contém**: Verifica se o texto contém uma parte que corresponde à expressão regular;
 - **Extrai**: Extrai do texto somente a parte coincidente com a expressão regular;
 - **Exato**: Verifica se o texto todo é exatamente coincidente com a expressão regular;
 - **Todas**: Extrai do texto todas as coincidências com a expressão regular.
- **E** - Salva em arquivo texto as expressões contidas no grid (**G**);
- **F** - Abre PDF contendo os símbolos usados em expressões regulares;
- **G** - Grid contendo expressões salvas em arquivo texto.

Utilize o seguinte código para criar o evento **Click** do botão **Contém**:

```
private void btnContem_Click(object sender, EventArgs e)
{
    // define sem nenhuma opção de teste
    RegexOptions opt = RegexOptions.None;
    // se o checkBox estiver selecionado, não diferencia
    // minúsculas de maiúsculas
    if (ckbIgnoreCase.Checked) opt = RegexOptions.IgnoreCase;
    // cria instância de Regex
    Regex regex = new Regex(tbxExprReg.Text, opt);
    // Testa se a string é compatível com a expressão
    if (regex.IsMatch(tbxTeste.Text))
        lblResultado.Text = "Ok";
    else
        lblResultado.Text = "Falha";
}
```

Utilize o seguinte código para criar o evento **Click** do botão **Extrai**:

```
private void btnExtrai_Click(object sender, EventArgs e)
{
    // define sem nenhuma opção de teste
    RegexOptions opt = RegexOptions.None;
    // se o checkBox estiver selecionado, não diferencia
    // minúsculas de maiúsculas
    if (ckbIgnoreCase.Checked) opt = RegexOptions.IgnoreCase;
    // cria instância de Regex
    Regex regex = new Regex("[A-Z]{3}-[0-9]{4}", opt);
    // cria objeto contendo os resultados da expressão
    Match m = regex.Match(tbxTeste.Text);
    // pega a posição e o tamanho do texto compatível
    int ini = m.Index;
    int tam = m.Length;
    // se existe compatibilidade, extrai o texto compatível
    if (m.Success)
        //lblResultado.Text = tbxTeste.Text.Substring(ini,tam);
        lblResultado.Text = m.Value;

    else
        lblResultado.Text = "Falha";
}
```

Utilize o seguinte código para criar o evento **Click** do botão **Exato**:

```
private void btnExato_Click(object sender, EventArgs e)
{
    RegexOptions opt = RegexOptions.None;
    if (ckbIgnoreCase.Checked) opt = RegexOptions.IgnoreCase;
    Regex regex = new Regex(tbxExprReg.Text, opt);
    Match m = regex.Match(tbxTeste.Text);
    int ini = m.Index;
    int tam = m.Length;
    if (m.Success)
    {
```

C# - Módulo I

```
// verifica se o texto é exatamente compatível com a expressão.  
if (tbxTeste.Text.Length == tam)  
    lblResultado.Text = "Exato";  
else  
{  
    lblResultado.Text = "Contém";  
}  
else  
    lblResultado.Text = "Falha";  
}
```

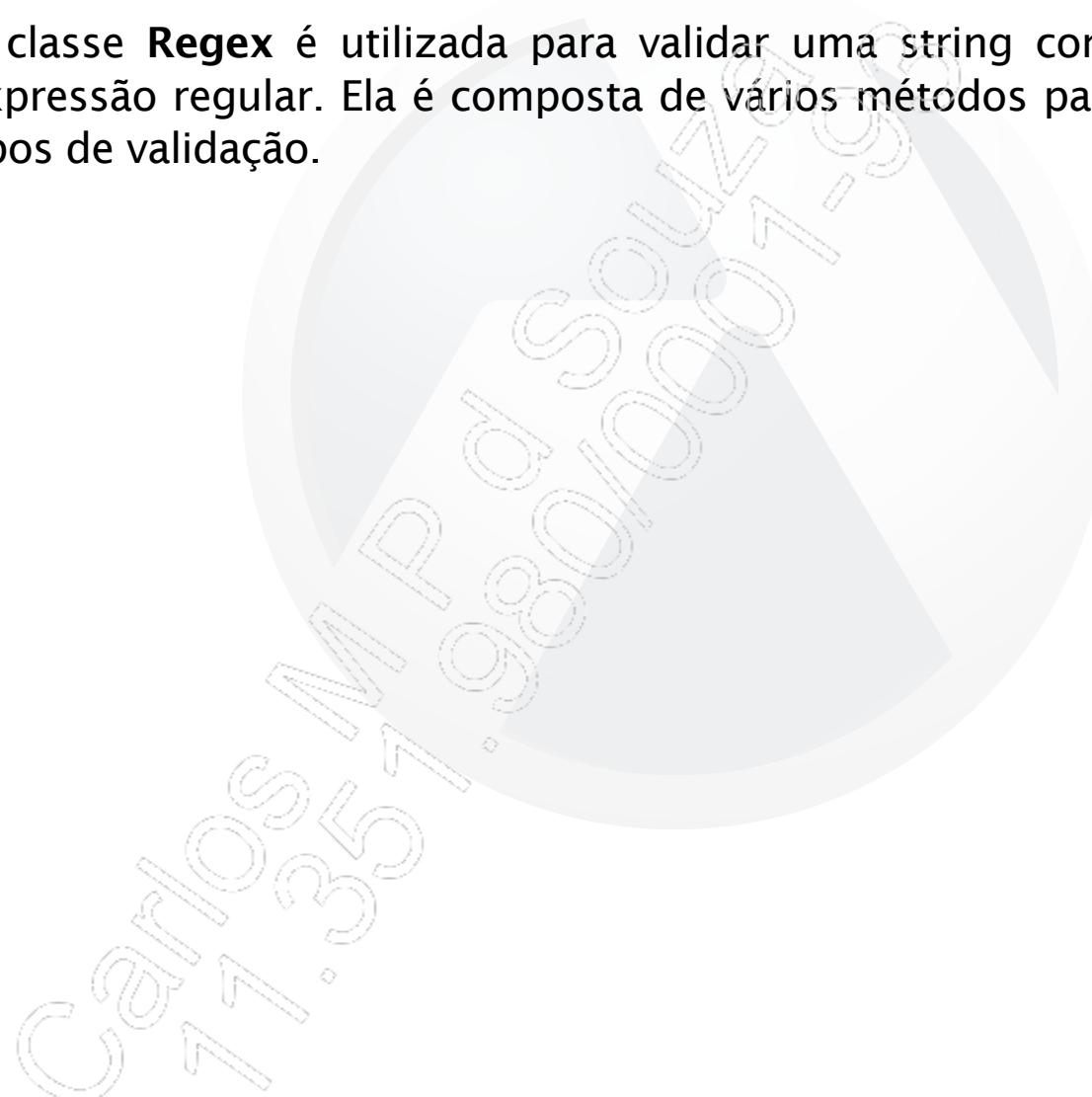
Utilize o seguinte código para criar o evento **Click** do botão **Todas**:

```
private void btnTodas_Click(object sender, EventArgs e)  
{  
    RegexOptions opt = RegexOptions.None;  
    if (ckbIgnoreCase.Checked) opt = RegexOptions.IgnoreCase;  
    Regex regex = new Regex(tbxExprReg.Text, opt);  
  
    // recupera todos os trechos do texto compatíveis com a expressão  
    MatchCollection ms = regex.Matches(tbxTeste.Text);  
    // se existir alguma coincidência  
    if (ms.Count > 0)  
    {  
        // mostra todas elas na área de resposta  
        lblResultado.Text = "";  
        foreach (Match m in ms)  
            lblResultado.Text += m.Value + ";" ;  
    }  
    else  
        lblResultado.Text = "Falha";  
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Uma expressão regular é definida como uma string formada por símbolos especiais que nos permite validar se um dado está no formato correto;
- Os principais símbolos utilizados em expressões regulares são: . (ponto), \$, ^, *, +, ?, [], |, (), {n} e {de, até};
- A classe **Regex** é utilizada para validar uma string com base em uma expressão regular. Ela é composta de vários métodos para fazer diversos tipos de validação.



10

Expressões regulares

Teste seus conhecimentos

Carlos M. 77.357.900-003
SOUZA



IMPACTA
EDITORA

1. Qual expressão regular melhor representa um teste que verifica se o texto contém um CEP (Código de Endereçamento Postal)?

- a) [A-Z]{3}-[0-9]{4}
- b) [0-9]{5}-[0-9]{3}
- c) [0-9]{3}\.[0-9]{3}\.[0-9]{3}\-[0-9]{2}
- d) ^[A-Z]{3}-[0-9]{4}\$
- e) ^[0-9]{5}-[0-9]{3}\$

2. Qual expressão regular melhor representa um teste que verifica se o texto contém um CPF (Cadastro de Pessoa Física)?

- a) [A-Z]{3}-[0-9]{4}
- b) [0-9]{5}-[0-9]{3}
- c) [0-9]{3}\.[0-9]{3}\.[0-9]{3}\-[0-9]{2}
- d) ^[A-Z]{3}-[0-9]{4}\$
- e) ^[0-9]{5}-[0-9]{3}\$

3. Qual expressão regular melhor representa um teste que verifica se o texto é exatamente igual a um CEP?

- a) [A-Z]{3}-[0-9]{4}
- b) [0-9]{5}-[0-9]{3}
- c) [0-9]{3}\.[0-9]{3}\.[0-9]{3}\-[0-9]{2}
- d) ^[A-Z]{3}-[0-9]{4}\$
- e) ^[0-9]{5}-[0-9]{3}\$

4. Qual método é usado para extraímos todas as correspondências que o texto tem com a expressão regular?

- a) IsMatch()
- b) IsMatches()
- c) Match()
- d) Matches()
- e) MatchCollection

5. Qual método booleano é usado para indicar se o texto possui alguma correspondência com a expressão regular?

- a) IsMatch()
- b) IsMatches()
- c) Match()
- d) Matches()
- e) MatchCollection

Coleções

11

- ✓ Diferenças entre coleções e arrays;
- ✓ ArrayList;
- ✓ Classe Stack;
- ✓ Classe Queue;
- ✓ List<tipo>;
- ✓ Inicializadores de List.



IMPACTA
EDITORA

11.1. Introdução

Coleções são estruturas oferecidas pelo .NET Framework para coletar elementos de qualquer tipo. As classes **Collection** podem ser utilizadas no lugar dos arrays, os quais também recolhem elementos, mas apresentam algumas limitações.

O tipo de elemento de uma classe **Collection** é um objeto. Por essa razão, os elementos pertencentes a uma classe **Collection** básica são aceitos como objetos, bem como retornados e mantidos como tais.

Ao introduzirmos um valor em uma coleção, este sofre a operação de boxing (se for um tipo-valor, é transformado em um tipo-referência). Porém, devemos aplicar a operação unboxing (transformar em referência) ao valor, se posteriormente quisermos acessar seus dados.

No caso de um array de objetos, podemos adicionar valores de qualquer tipo nele. Se o tipo do valor for um inteiro, teremos o seguinte:

- O valor é submetido automaticamente ao boxing;
- A cópia desse valor inteiro introduzido é referenciada pelo elemento do array, o qual é uma referência de objeto.



As classes **Collection** são mantidas no namespace **System.Collections** e em **System.Collections.Generics**.

11.2. Diferenças entre coleções e arrays

A tabela adiante descreve as principais diferenças existentes entre arrays e coleções:

Característica	Coleções	Arrays
Redimensionamento	Têm a capacidade de se redimensionar dinamicamente conforme necessário.	Não podem aumentar ou diminuir de tamanho. O array possui tamanho fixo.
Estrutura de dados	Os itens dentro de um loop são disponibilizados como somente leitura.	Não há modo de criar arrays somente leitura. Eles são uma estrutura de dados de leitura e gravação.

11.3. ArrayList

Antes de verificarmos a funcionalidade da classe **ArrayList**, devemos considerar alguns aspectos acerca dos arrays comuns, pois sua utilização possui algumas implicações:

- Ao remover um elemento do array, há a necessidade de copiar e mover para cima todos os elementos seguintes, o que resulta em duas cópias do último elemento;
- Para redimensionar um array, é necessário criar um novo array, copiar os elementos (se o array for menor, alguns elementos não serão incluídos) e atualizar as referências do array;
- Para incluir um elemento no array, é preciso mover os elementos para baixo, a fim de liberar um slot, o que faz com que o último elemento do array seja perdido.

Como podemos perceber, os arrays, apesar de úteis, apresentam algumas restrições em sua utilização. Para casos em que o número de elementos varia, a melhor solução é o uso de coleções. Uma das classes do .NET disponíveis para gerenciar uma coleção é a classe **ArrayList**, localizada dentro do namespace **System.Collections**:

```
using System.Collections;
```

C# - Módulo I

Para instanciarmos um **ArrayList**, utilizamos o código a seguir:

```
public partial class Form1 : Form
{
    // gerador de números aleatórios
    Random rn = new Random();

    public Form1()
    {
        InitializeComponent();
    }

    // é um array de objetos que tem seus próprios métodos para
    // inserir, adicionar, procurar, ordenar etc...
    /*
     * Métodos Importantes:
     *      Add(objeto)
     *      Insert(posicao, objeto)
     *      IndexOf(objeto)
     *      RemoveAt(posicao)
     *      Sort()
     */
    ArrayList arrList = new ArrayList();
}

/// <summary>
/// Mostra todos os elementos do ArrayList em um ListBox
/// </summary>
void mostraArrayList()
{
    lbxAL.Items.Clear();
    for (int i = 0; i < arrList.Count; i++)
    {
        lbxAL.Items.Add(arrList[i]);
    }
}
```

Por meio do método **Add**, é possível adicionar um elemento ao final de um **ArrayList**, o qual é redimensionado automaticamente:

```
private void btnAddAL_Click(object sender, EventArgs e)
{
    arrList.Add(rn.Next(100));
    mostraArrayList();
}
```

O método **Insert(posição, item)** insere um novo item na posição selecionada:

```
private void btnInfoALIns_Click(object sender, EventArgs e)
{
    // posição do item selecionado no ListBox
    int pos = lbxAL.SelectedIndex;
    // se existir algum item selecionado
    if (pos >= 0)
    {
        // insere o novo item nesta posição
        arrList.Insert(pos, rn.Next(100));
        mostraArrayList();
    }
    else MessageBox.Show("Nada selecionado");
}
```

Por meio do método **RemoveAt(posição)**, podemos remover um elemento de um **ArrayList** que está na posição indicada:

```
private void btnALExclui_Click(object sender, EventArgs e)
{
    int pos = lbxAL.SelectedIndex;
    if (pos >= 0)
    {
        arrList.RemoveAt(pos);
        mostraArrayList();
    }
    else MessageBox.Show("Nada selecionado");
}
```

O método **IndexOf(objeto)** procura o objeto nos itens do **ArrayList** retornando com a posição onde ele foi encontrado ou -1 caso não exista:

```
private void btnProcuraAL_Click(object sender, EventArgs e)
{
    // procura o conteúdo do TextBox no ArrayList
    int pos = arrList.IndexOf(Convert.ToInt32(tbxProcuraAL.Text));
    // se a posição for maior ou igual a zero, seleciona
    // no ListBox o item que está na mesma posição
    if (pos >= 0) lbxAL.SelectedIndex = pos;
    else MessageBox.Show("Não encontrado");
}
```

C# - Módulo I

Para ordenar os itens da lista, existe um método chamado **Sort()**, que faz uma ordenação padrão em ordem ascendente. Para alterar a forma da ordenação, descendente, por exemplo, precisamos implementar uma interface chamada **IComparer**. Ela possui um único método chamado **Compare** que recebe dois elementos da lista e deve retornar um número inteiro indicando em que ordem eles deverão ficar:

```
// classe para definir ordenação na descendente
class ordenaArrayList : IComparer
{
    // recebe 2 elementos da lista e retorna um número inteiro que pode
    // ser:
    // -1: ordena x antes de y
    // 0: mantém como está
    // 1: ordena x depois de y
    public int Compare(object x, object y)
    {
        // se x for menor que y, ordenar x depois de y
        if ((int)x < (int)y) return 1;
        // se x for maior que y, ordenar x antes de y
        else if ((int)x > (int)y) return -1;
        // se forem iguais, manter
        else return 0;
    }
}
```

Agora vamos usar este código no botão que faz a ordenação:

```
private void btnALSort_Click(object sender, EventArgs e)
{
    // se for ordenação na ascendente, usar o método Sort() normal
    if (rbAsc.Checked) arrList.Sort();
    // se for na descendente, passar para o método Sort()
    // uma instância da classe ordenaArrayList
    else arrList.Sort(new ordenaArrayList());
    // mostrar o ArrayList já ordenado
    mostraArrayList();
}
```

A lista a seguir exibe e descreve as principais propriedades e métodos da classe **ArrayList**:

Propriedade ou método	Descrição
Propriedade Count	Retorna o número de elementos.
Método Add	Adiciona um item no final da coleção.
Método Sort	Ordena os itens.
Método Contains	Retorna o índice de um elemento se ele estiver na coleção.
Método IndexOf	Igual ao Contains , retorna o índice de um elemento.
Método Clone	Duplica o ArrayList .
Método LastIndexOf	Retorna a última ocorrência de um item em um array.
Método Insert	Insere um elemento no meio da coleção.

11.4. Classe Stack

Esta classe permite a inclusão de elementos que são colocados em uma pilha. O elemento junta-se à pilha no topo e sai dela também a partir do topo. A classe **Stack** utiliza o sistema LIFO (last-in first-out), ou seja, o último a entrar é o primeiro a sair). Esta classe não possui indexador, ou seja, não é possível acessar seus elementos pela posição onde estão.

Os principais métodos são:

- **Push**: Coloca um item na pilha;
- **Pop**: Retira o último item da pilha e retorna uma referência a este objeto;
- **Peek**: Retorna o objeto que está no topo da pilha.

Vejamos um exemplo:

```
// é uma lista de objetos do tipo “pilha de pratos”
// Último que entra é o primeiro que sai
// LIFO (Last-in first-out)
// não possui indexador
```

C# - Módulo I

```
Stack stk = new Stack();
// Generics: pilha com elementos tipados
Stack<int> stkInt = new Stack<int>();
/*
 * Adicionar elemento no topo da pilha
 *     stack.Push( objeto );
 * Retirar elemento do topo da pilha
 *     stack.Pop();
 * Consultar elemento no topo da pilha
 *     stack.Peek();
 */
```

Utilizamos o método a seguir para exibir os elementos da pilha em um **ListBox**:

```
/// <summary>
/// Exibe o conteúdo da pilha no ListBox
/// </summary>
void mostraPilha()
{
    lbxStack.Items.Clear();
    // classe Stack não possui indexador, não dá para
    // percorrer com "for", tem que ser "foreach"
    foreach (object item in stk)
    {
        lbxStack.Items.Add(item);
    }
}
```

Este código permite incluir o elemento no topo da pilha:

```
// Inclui elemento no topo da pilha
private void btnPush_Click(object sender, EventArgs e)
{
    stk.Push(rn.Next(100));
    // mostra o conteúdo da pilha no ListBox
    mostraPilha();
}
```

Para retirar o elemento que está no topo da pilha e exibi-lo, utilizamos o código a seguir;

```
private void btnPop_Click(object sender, EventArgs e)
{
    // se tentarmos retirar um elemento da pilha e ela
    // estiver vazia, ocorrerá um erro
    if (stk.Count > 0)
    {
        // retira o primeiro elemento da pilha e retorna
        // com o seu conteúdo
        lblPeekStack.Text = stk.Pop().ToString();
        // mostra o novo conteúdo da pilha, já sem seu primeiro
        // elemento
        mostraPilha();
    }
}
```

Utilizamos este código para mostrar o elemento que está no topo da pilha sem retirá-lo dela:

```
private void btnPeek_Click(object sender, EventArgs e)
{
    if (stk.Count > 0)
    {
        // retorna com o conteúdo do primeiro elemento
        // sem retirá-lo da pilha
        lblPeekStack.Text = stk.Peek().ToString();
        mostraPilha();
    }
}
```

 É necessária a diretiva **using System.Collections**. Caso usemos **Stack<tipo>**, será necessário **System.Collections.Generics**;

11.5. Classe Queue

Nesta classe, o elemento é inserido na parte de trás da fila. Esta operação é chamada de enfileiramento. Este mesmo elemento sai a partir da frente da fila, operação chamada desenfileiramento. Esse procedimento de entrada e saída recebe o nome de FIFO (first-in first-out, isto é, o primeiro a entrar é o primeiro a sair). Os principais métodos dessa classe, que também não possui indexador, são:

- **Enqueue**: Coloca um item na fila;
- **Dequeue**: Retira o primeiro item da fila e retorna uma referência;
- **Peek**: Retorna o primeiro item.

Vejamos um exemplo:

```
// lista de objetos do tipo “fila”
// FIFO - First-in first-out
// não possui indexador
Queue que = new Queue();
// Generics: Fila com elementos tipados
Queue<string> que2 = new Queue<string>();
/*
 * Adiciona elemento no final da fila
 * Queue.Enqueue( elemento );
 * Remove elemento do início da fila
 * Queue.Dequeue();
 * Retorna com o primeiro da fila
 * Queue.Peek()
*/
```

Utilizamos este método para exibir o conteúdo da fila em um ListBox:

```
void mostraFila()
{
    lbxQueue.Items.Clear();
    // não possui indexador, tem que usar foreach
    foreach (object item in que)
    {
        lbxQueue.Items.Add(item);
    }
}
```

Este método permite incluir o elemento no final da fila:

```
private void btnQueEnq_Click(object sender, EventArgs e)
{
    // insere elemento no fim da fila
    que.Enqueue(rn.Next(100));
    // mostra a lista com o novo elemento no final
    mostraFila();
}
```

Este código permite retirar o primeiro elemento da fila:

```
private void btnQueDeq_Click(object sender, EventArgs e)
{
    if (que.Count > 0)
    {
        lblPeekQueue.Text = que.Dequeue().ToString();
        mostraFila();
    }
}
```

Por meio deste código, é possível consultar o primeiro elemento da fila:

```
private void btnQuePeek_Click(object sender, EventArgs e)
{
    if (que.Count > 0)
    {
        lblPeekQueue.Text = que.Peek().ToString();
        mostraFila();
    }
}
```

! É necessária a diretiva `using System.Collections`. Caso usemos `Queue<tipo>`, precisaremos de `System.Collections.Generics`;

11.6. List<tipo>

Um **List<tipo>** é muito semelhante a um **ArrayList**, mas com tipo de dado definido. Qualquer tipo de dado pode fazer parte da lista. Vejamos um exemplo:

```
/*
 * List: É uma coleção na qual podemos definir o tipo de dado dos elementos
 *
 * Exemplo:
 *
 *     List<int> dados = new List<int>();
 *     List<DateTime> lista = new List<DateTime>();
 *
 *     class Aluno
 *     {
 *         public int Código {get; set;}
 *         public string Nome {get; set;}
 *         public string EMail {get; set;}
 *         public DateTime DataNasc {get; set;}
 *     }
 *
 *     List<Aluno> tbx = new List<Aluno>();
 *
 * Métodos Principais
 *
 *     Add(elemento),
 *     Insert(posição, elemento),
 *     IndexOf(valor),
 *     RemoveAt(posição)
 *     Sort()
 *
 */
List<int> lst = new List<int>();
```

Este método permite mostrar a lista no ListBox:

```
void mostraListInt()
{
    lbxLst.Items.Clear();
    foreach (int n in lst)
    {
        lbxLst.Items.Add(n);
    }
}
```

Com este código, podemos adicionar um número aleatório na lista:

```
private void btnLstIntAdd_Click(object sender, EventArgs e)
{
    lst.Add(rn.Next(100));
    mostraListInt();
}
```

Este código permite inserir um elemento na posição indicada:

```
private void btnLstIntIns_Click(object sender, EventArgs e)
{
    // 1. descobrir a posição do item selecionado no ListBox
    int pos = lbxLst.SelectedIndex;
    // 2. se for uma posição válida (maior ou igual a zero)
    if (pos >= 0)
    {
        // 2.1. Usando o método Insert, inserir número
        //       randômico nesta posição
        lst.Insert(pos, rn.Next(100));
        // 2.2. Mostrar a lista no listBox
        mostraListInt();
    }
}
```

Para procurar o dado digitado no TextBox na lista, utilizamos este código:

```
private void btnLstIntProcura_Click(object sender, EventArgs e)
{
    // 1. converter o dado digitado em tbxProcuraLst para
    //     inteiro e armazenar em variável
    int numero = Convert.ToInt32(tbxProcuraLst.Text);
    // 2. usando o método IndexOf(), procurar o número na lista
    int pos = lst.IndexOf(numero);
    // 3. se o valor retornado por IndexOf() (posição)
    //     for maior ou igual a zero, selecionar no listBox
    //     o item que está na mesma posição
    if (pos >= 0)
        lbxLst.SelectedIndex = pos;
    else
        MessageBox.Show("Não encontrado");
}
```

C# - Módulo I

Para remover um elemento da lista, utilizamos o seguinte código:

```
private void btnLstIntExclui_Click(object sender, EventArgs e)
{
    // 1. posição do item selecionado no ListBox, que é
    //     o elemento que iremos excluir
    int pos = lbxLst.SelectedIndex;
    // 2. se for uma posição válida, excluir
    if (pos >= 0)
    {
        lst.RemoveAt(pos);
        mostraListInt();
    }
    else MessageBox.Show("Nenhum item selecionado");
}
```

O código adiante permite ordenar os dados da lista. Note que este exemplo é muito semelhante ao que vimos no **ArrayList**, exceto pelo método **Compare** da interface **IComparer** que recebe x e y tipados:

```
class ordenaListInt : IComparer<int>
{
    public int Compare(int x, int y)
    {
        if (x < y) return 1;
        else if (x > y) return -1;
        else return 0;
    }
}

private void btnLstIntOrdena_Click(object sender, EventArgs e)
{
    if (rbAsc2.Checked) lst.Sort();
    else lst.Sort(new ordenaListInt());
    mostraListInt();
}
```

11.7. Inicializadores de List

Com uma sintaxe semelhante à que utilizamos para arrays, é possível inicializar uma coleção no momento de sua declaração. Vejamos o exemplo a seguir:

```
List<int> lst = new List<int> { 4,8,2,19,28,33,10 };
List<DateTime> lstData = new List<DateTime> { new DateTime(2013, 2, 25),
                                              new DateTime(2013, 3, 30),
                                              new DateTime(2013, 4, 15),
                                              new DateTime(2013, 5, 10) };

List<Deptos> deptos = new List<Deptos>
{
    new Deptos { COD_DEPTO = 1, DEPTO = "PESSOAL" },
    new Deptos { COD_DEPTO = 2, DEPTO = "C.P.D." },
    new Deptos { COD_DEPTO = 3, DEPTO = "CONTROLE DE ESTOQUE" },
    new Deptos { COD_DEPTO = 4, DEPTO = "COMPRAS" },
    new Deptos { COD_DEPTO = 5, DEPTO = "PRODUCAO" },
    new Deptos { COD_DEPTO = 6, DEPTO = "DIRETORIA" },
    new Deptos { COD_DEPTO = 7, DEPTO = "TELEMARKETING" },
    new Deptos { COD_DEPTO = 8, DEPTO = "FINANCIERO" },
    new Deptos { COD_DEPTO = 9, DEPTO = "RECURSOS HUMANOS" },
    new Deptos { COD_DEPTO = 10, DEPTO = "TREINAMENTO" },
    new Deptos { COD_DEPTO = 11, DEPTO = "PRESIDENCIA" },
    new Deptos { COD_DEPTO = 12, DEPTO = "PORTARIA" },
    new Deptos { COD_DEPTO = 13, DEPTO = "CONTROLADORIA" },
    new Deptos { COD_DEPTO = 14, DEPTO = "P.C.P." }
};

class Deptos
{
    public int COD_DEPTO;
    public string DEPTO;
}
```

! É necessária a diretiva
`using System.Collections.Generics;`

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- O .NET Framework oferece estruturas chamadas coleções, que servem para coletar elementos de qualquer tipo. As classes **Collection** podem ser utilizadas no lugar dos arrays, os quais também recolhem elementos, mas apresentam algumas limitações;
- A classe **ArrayList** é uma das classes do .NET disponíveis para gerenciar uma coleção. Ela está localizada dentro do namespace **System.Collections**;
- A classe **Stack** permite a inclusão de elementos que são colocados em uma pilha. Seus principais métodos são **Push**, **Pop** e **Peek**;
- Na classe **Queue**, ocorre a operação de enfileiramento, isto é, o elemento é inserido na parte de trás da fila. Os principais métodos dessa classe são **Enqueue**, **Dequeue** e **Peek**.

11

Coleções

Teste seus conhecimentos

Carlos M. 77.357.900/0007-93
SOUZA



IMPACTA
EDITORA

1. Qual o método que adiciona um elemento no topo de um objeto do tipo Stack?

- a) Pop()
- b) Enqueue()
- c) Add
- d) Push()
- e) Não há como adicionar um elemento no topo de um Stack.

2. Qual o método que adiciona um elemento no final (último) de um objeto do tipo Stack?

- a) Pop()
- b) Enqueue()
- c) Add
- d) Push()
- e) Não há como adicionar um elemento no final de um Stack.

3. Qual o método que adiciona um elemento no topo de um objeto do tipo Queue?

- a) Pop()
- b) Enqueue()
- c) Add
- d) Push()
- e) Não há como adicionar um elemento no topo de um Queue.

4. Qual o método que adiciona um elemento no final (último) de um objeto do tipo Queue?

- a) Pop()
- b) Enqueue()
- c) Add
- d) Push()
- e) Não há como adicionar um elemento no final de um Queue.

5. Assinale a alternativa que completa adequadamente a frase a seguir:

Para ordenar um objeto List de modo diferente do padrão, precisamos implementar uma interface chamada _____ e o seu método _____.

- a) Comparer, Comparer.
- b) Compare, IComparer.
- c) IComparer, Sort().
- d) IComparer, Compare().
- e) Não há como ordenar um objeto List.

Acesso à API do Windows

12

- ✓ Exemplo: Windows DLL.

Carlos M. P. Souza
77.357.980/0007-03



IMPACTA
EDITORA

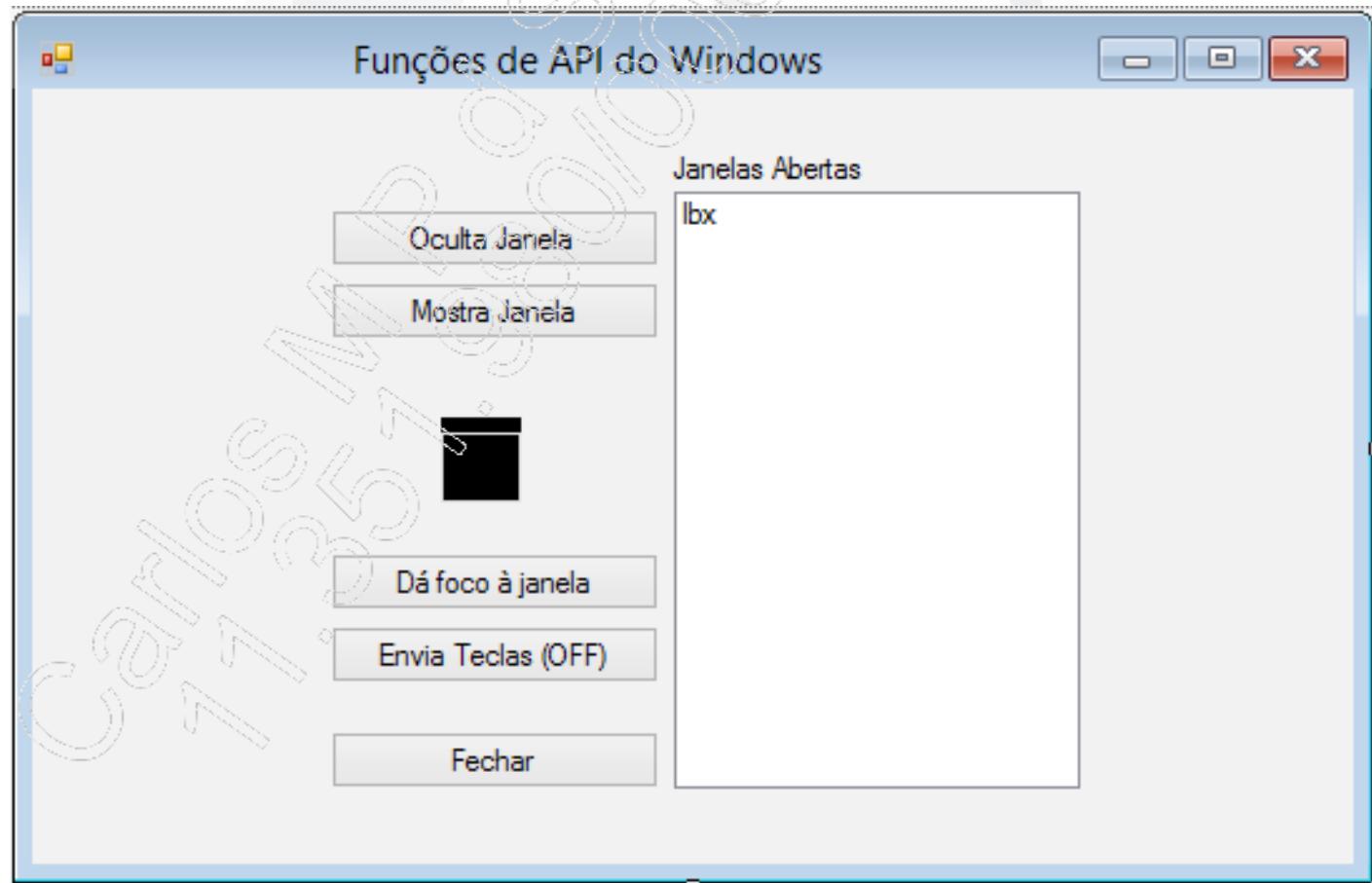
12.1. Introdução

Toda a programação para Windows se baseia em um conjunto de funções existentes no sistema operacional. Essas funções são chamadas de API (Application Programming Interface) do Windows e estão contidas em bibliotecas de acesso dinâmico (DLLs).

Para podermos utilizar uma função API do Windows, precisamos saber em que DLL ela está e também a sua sintaxe, ou seja, os parâmetros que ela recebe e o valor que retorna.

12.2. Exemplo: Windows DLL

Neste exemplo, vamos trabalhar com algumas funções de API do Windows:



12.2.1. Arredondando as bordas de um formulário

Neste exemplo, vamos aprender a arredondar as bordas de um formulário:

1. Declare os namespaces necessários;

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

// para usar DllImport
using System.Runtime.InteropServices;
// para usar a classe Process
using System.Diagnostics;
```

```
namespace WindowsDLL
{
    public partial class Form1 : Form
    {
```

2. Declare a função do Windows para gerar uma região elíptica;

```
// para usar a classe Process
using System.Diagnostics;

namespace WindowsDLL
{
    public partial class Form1 : Form
    {
        // define uma região elíptica para ser usada por um controle
        [DllImport("Gdi32.dll", EntryPoint = "CreateRoundRectRgn")]
        private static extern IntPtr CreateRoundRectRgn
        (
            int nLeftRect, // coordenada x do canto superior esquerdo
            int nTopRect, // coordenada y do canto superior esquerdo
            int nRightRect, // coordenada x do canto inferior direito
            int nBottomRect, // coordenada y do canto inferior direito
            int nWidthEllipse, // altura da elipse
            int nHeightEllipse // largura da elipse
        );
```

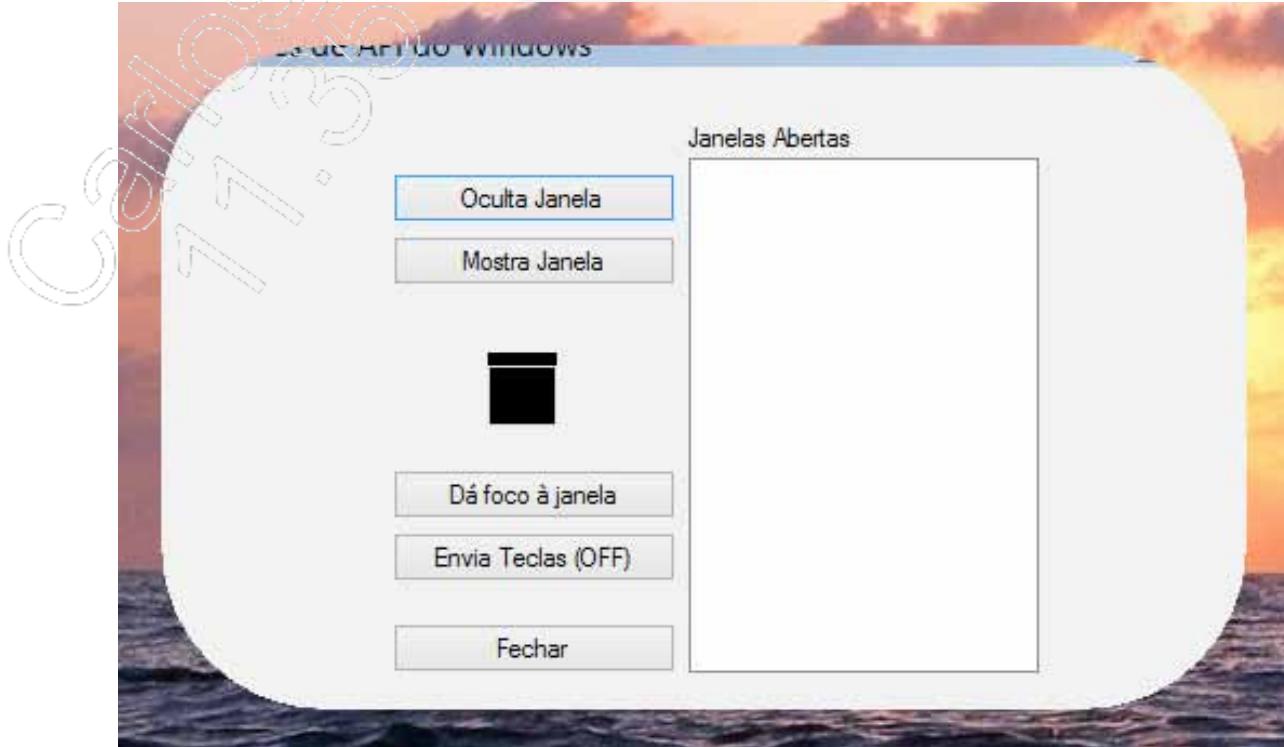
C# - Módulo I

3. Altere o método construtor de **Form1** incluindo o código mostrado;

```
namespace WindowsDLL
{
    public partial class Form1 : Form
    {
        // define uma região elíptica para ser usada por um controle
        [DllImport("Gdi32.dll", EntryPoint = "CreateRoundRectRgn")]
        private static extern IntPtr CreateRoundRectRgn
        (
            int nLeftRect, // coordenada x do canto superior esquerdo
            int nTopRect, // coordenada y do canto superior esquerdo
            int nRightRect, // coordenada x do canto inferior direito
            int nBottomRect, // coordenada y do canto inferior direito
            int nWidthEllipse, // altura da elipse
            int nHeightEllipse // largura da elipse
        );

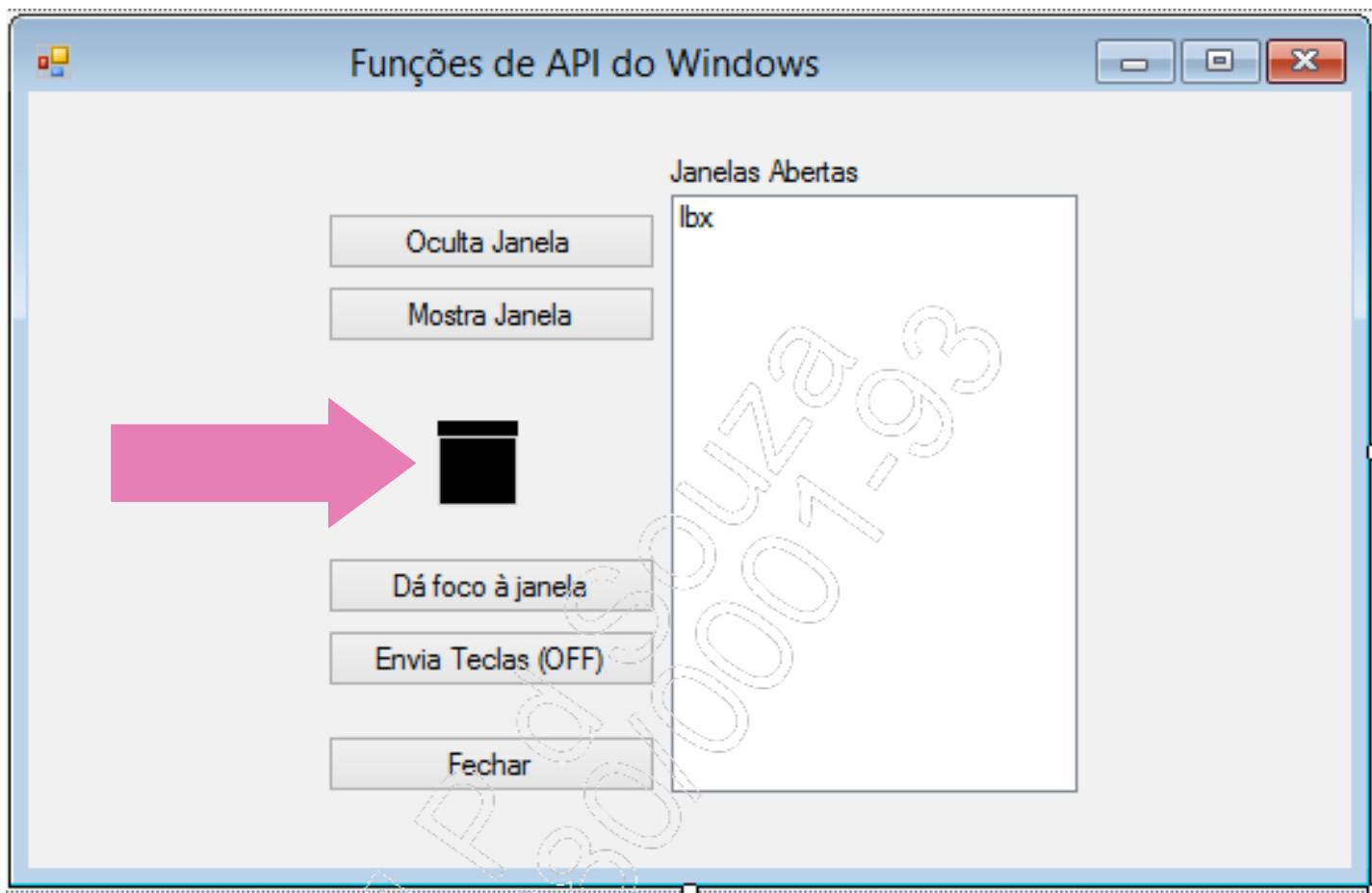
        public Form1()
        {
            InitializeComponent();
            // Altera a propriedade Region do Form tornando-o elíptico
            this.Region = Region.FromHrgn(CreateRoundRectRgn(20, 20,
                Width - 20, Height - 20, 150, 150));
        }
    }
}
```

4. Execute e observe como ficou o formulário.



12.2.2. Utilizando algumas funções de API

Neste formulário, existe um GroupBox de fundo preto chamado **gb**, indicado a seguir:



O objetivo é movê-lo usando as setas de movimentação, independente de qual controle tenha foco, inclusive se o foco estiver em outra aplicação. Neste caso, não podemos usar o evento **KeyDown** do Form, porque, por exemplo, se pressionarmos a seta para cima junto com seta para a direita, desejamos que o GroupBox se move na diagonal. Além disso, neste exemplo, vamos aprender a mostrar, ocultar e dar foco a uma janela e enviar teclas para a aplicação.

C# - Módulo I

1. Declare a função do Windows para verificar o status de uma tecla;

```
// Retorna com um inteiro de 16 bits indicando o status de uma tecla  
// 10000000 00000000: tecla está pressionada  
// 00000000 00000000: tecla está solta  
// 00000000 00000001: tecla está ligada (Caps, Num)  
// 00000000 00000000: tecla está desligada  
[DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling = true)]  
private static extern short GetKeyState(int keyCode);
```

```
public Form1()  
{  
    InitializeComponent();  
    // Altera a propriedade Region do Form tornando-o elíptico  
    this.Region = Region.FromHrgn(CreateRoundRectRgn(20, 20,  
        Width - 20, Height - 20, 150, 150));  
}
```

2. O formulário deve ter um timer (timer1) com propriedade **Enabled** igual a **true** e **Interval** igual a 100. O evento **Tick** desse timer vai verificar o status de cada tecla de movimentação e, se ela estiver pressionada, vai mover o objeto para aquela direção. Como os IFs são independentes, se pressionarmos mais de uma tecla, ambas serão testadas;

O bit de maior ordem (o primeiro do lado esquerdo) indica se a tecla está pressionada. Em um dado do tipo short, este bit indica o sinal do número. Se 1, o número é negativo.

```
private void timer1_Tick(object sender, EventArgs e)  
{  
    // se a tecla seta para baixo estiver pressionada  
    if (GetKeyState((int)Keys.Down) < 0)  
    {  
        // move o groupBox para baixo  
        gb.Top += 10;  
        if (gb.Top > this.Height) gb.Top = -gb.Height;  
    }  
}
```

```
// se a tecla seta para cima estiver pressionada
if (GetKeyState((int)Keys.Up) < 0)
{
    // move o groupBox para cima
    gb.Top -= 10;
    if (gb.Top < -gb.Height) gb.Top = this.Height;
}

// se a tecla seta para a direita estiver pressionada
if (GetKeyState((int)Keys.Right) < 0)
{
    // mover para a direita
    gb.Left += 10;
    if (gb.Left > this.Width) gb.Left = -gb.Width;
}

// se seta para a esquerda estiver pressionada
if (GetKeyState((int)Keys.Left) < 0)
{
    // mover para a esquerda
    gb.Left -= 10;
    if (gb.Left < -gb.Width) gb.Left = this.Width;
}

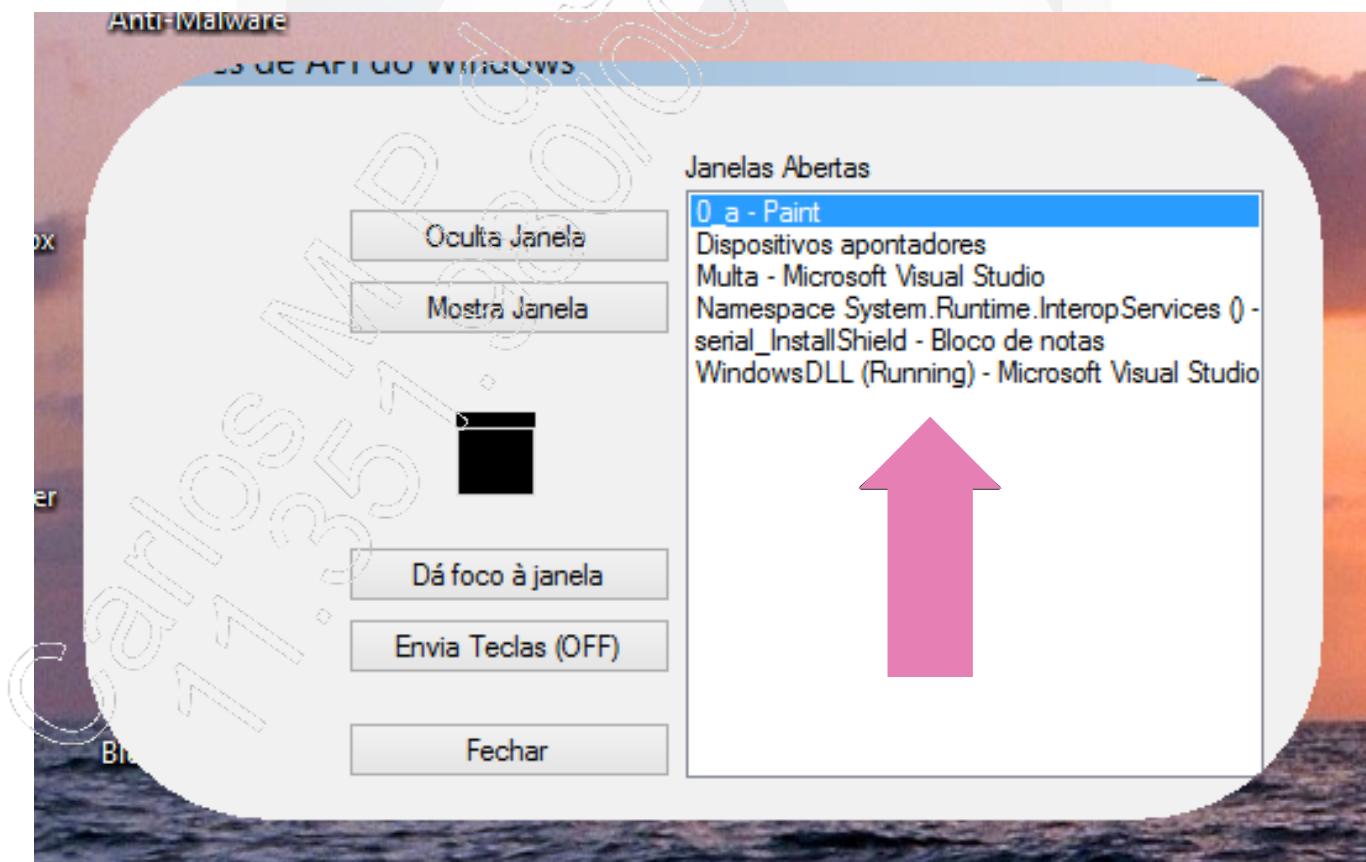
// muda a cor de fundo do groupBox de acordo com a tecla pressionada
if (GetKeyState((int)Keys.V) < 0) gb.BackColor = Color.Red;
if (GetKeyState((int)Keys.P) < 0) gb.BackColor = Color.Black;
if (GetKeyState((int)Keys.A) < 0) gb.BackColor = Color.Yellow;
if (GetKeyState((int)Keys.Z) < 0) gb.BackColor = Color.Blue;
}
```

 Observe que mesmo se estivermos com foco no Bloco de Notas, por exemplo, o objeto se move quando pressionamos as setas.

C# - Módulo I

3. Crie o evento **FormLoad** do formulário para carregar no **ListBox** os títulos de todos os aplicativos abertos no momento;

```
private void Form1_Load(object sender, EventArgs e)
{
    // cria array para armazenar as aplicações abertas no momento
    Process[] processos = Process.GetProcesses();
    // loop para percorrer as aplicações
    foreach (Process proc in processos)
        // se existir uma janela principal com título
        if (proc.MainWindowTitle != "")
    {
        // inclui o título da janela principal no ListBox
        lbx.Items.Add(proc.MainWindowTitle);
    }
}
```

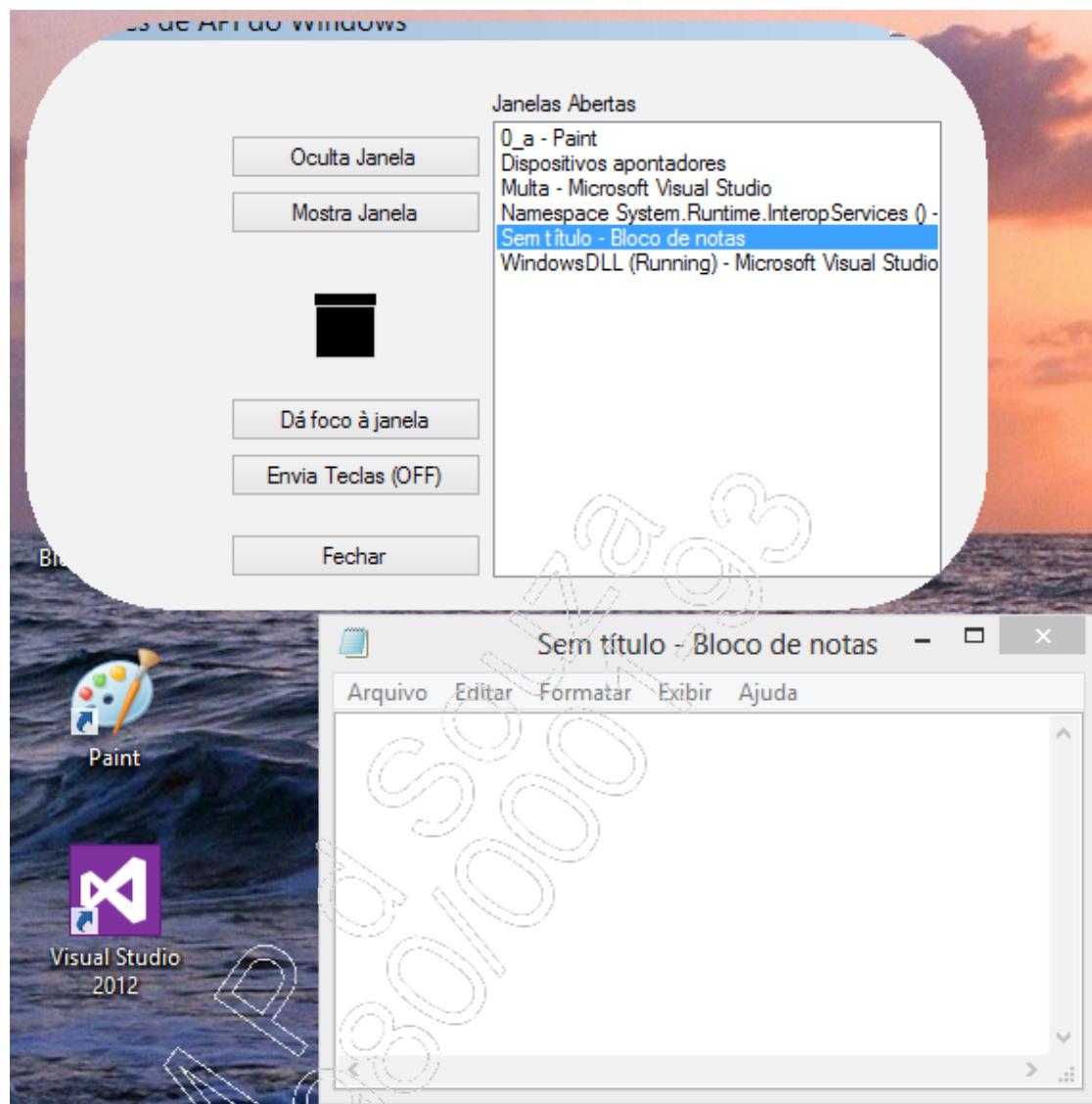


4. O botão **Oculta Janela** deve tornar invisível a janela que foi selecionada no ListBox, enquanto o **Mostra Janela** deve torná-la visível novamente. Para isso, é necessário declarar mais duas funções de API do Windows. Faça isso no mesmo local onde declarou as outras funções:

```
// retorna com o número identificador da janela a partir do nome  
// da classe registrada ou do seu título  
[DllImport("user32.dll")]  
private static extern int FindWindow(string className, string windowText);  
  
// torna uma janela visível ou invisível  
// hwnd: Identificador da janela  
// command: 0 para invisível e 1 para visível  
[DllImport("user32.dll")]  
private static extern int ShowWindow(int hwnd, int command);  
  
private void btnOcultar_Click(object sender, EventArgs e)  
{  
    // oculta a janela cujo título está selecionado no ListBox  
    int hwnd = FindWindow(null, lbx.SelectedItem.ToString());  
    ShowWindow(hwnd, SW_HIDE);  
}  
  
private void btnMostrar_Click(object sender, EventArgs e)  
{  
    // torna visível a janela cujo título está selecionado no ListBox  
    int hwnd = FindWindow(null, lbx.SelectedItem.ToString());  
    ShowWindow(hwnd, SW_SHOW);  
}
```

C# - Módulo I

5. Execute o aplicativo junto com algum outro. Na imagem a seguir, vemos o Bloco de Notas com um documento sem título;



6. Selecione o Bloco de Notas no ListBox e teste os botões **Oculta Janela** e **Mostra Janela**;

7. O próximo botão vai focar (trazer para a frente) o aplicativo selecionado no ListBox. Para isso, você precisará de mais uma função do Windows. Declare-a no início da classe **Form1** e depois crie o evento **Click** do botão;

```
// torna uma aplicação ativa. hWnd é o identificador da janela
[DllImport("USER32.DLL")]
public static extern bool SetForegroundWindow(IntPtr hWnd);
```

```
public Form1()
{
    InitializeComponent();
    // Altera a propriedade Region do Form tornando-o elíptico
    this.Region = Region.FromHrgn(CreateRoundRectRgn(20, 20,
        Width - 20, Height - 20, 150, 150));
}
```

```
// evento Click do botão btnDaFoco
private void btnDaFoco_Click(object sender, EventArgs e)
{
    // pega o identificador da janela cujo título está
    // selecionado no ListBox
    int hwnd = FindWindow(null, lbx.SelectedItem.ToString());
    // dá foco a esta janela
    SetForegroundWindow((IntPtr)hwnd);
}
```

8. O próximo botão vai ligar ou desligar um timer que vai enviar uma sequência de teclas para a aplicação que estiver com foco no momento. O timer fará o envio a cada 5 segundos, portanto, sua propriedade **Interval** deve ser configurada para 5000. Mantenha **Enabled** igual a **false**;

9. Declare a API do Windows para detectar qual janela tem o foco;

```
// retorna com o identificador da janela ativa no momento
[DllImport("USER32.DLL")]
public static extern IntPtr GetForegroundWindow();
```

10. Crie o evento **Click** do botão que liga ou desliga o timer:

```
private void btnEnviaTecla_Click(object sender, EventArgs e)
{
    // liga ou desliga o timer que envia teclas para
    // a aplicação com foco no momento
    timer2.Enabled = !timer2.Enabled;
    btnEnviaTecla.Text =
        timer2.Enabled ? "Envia Teclas (ON)" : "Envia Teclas (OFF)";
}
```

11. Crie o evento **Tick** do timer:

```
private void timer2_Tick(object sender, EventArgs e)
{
    // pega o identificador da janela (aplicação) ativa
    IntPtr appAtiva = GetForegroundWindow();
    // envia uma sequência de teclas para ela
    SendKeys.SendWait("IMPACTA TECNOLOGIA\n");
}
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Todos os recursos existentes no Windows são fornecidos por funções internas do sistema operacional;
- As funções internas do Windows estão em bibliotecas de acesso dinâmico (DLLs);
- Para declarar uma função do Windows, precisamos saber em que DLL ela está, seus parâmetros e seu valor de retorno.

Gerando Setup de instalação

13

- ✓ Instalação do InstallShield LE;
- ✓ Usando o InstallShield.

Carlos M. Souza
77.357.99000-0001-03



IMPACTA
EDITORA

13.1. Introdução

No Visual Studio 2012, restou apenas a opção **InstallShield Limited Edition**, que precisa ser instalada, já que a opção **Setup Project** não existe mais.

O **InstallShield** é um gerador de setup de instalação bastante conhecido que já existe há cerca de 10 anos. É comercializado pela empresa Flexera (www.flexerasoftware.com). Recentemente, foi criada a versão **InstallShield for Visual Studio 2012 LE (Limited Edition)** gratuita, que pode ser baixada pela Internet. Esta versão, no entanto, não possui todos os recursos existentes das versões pagas.

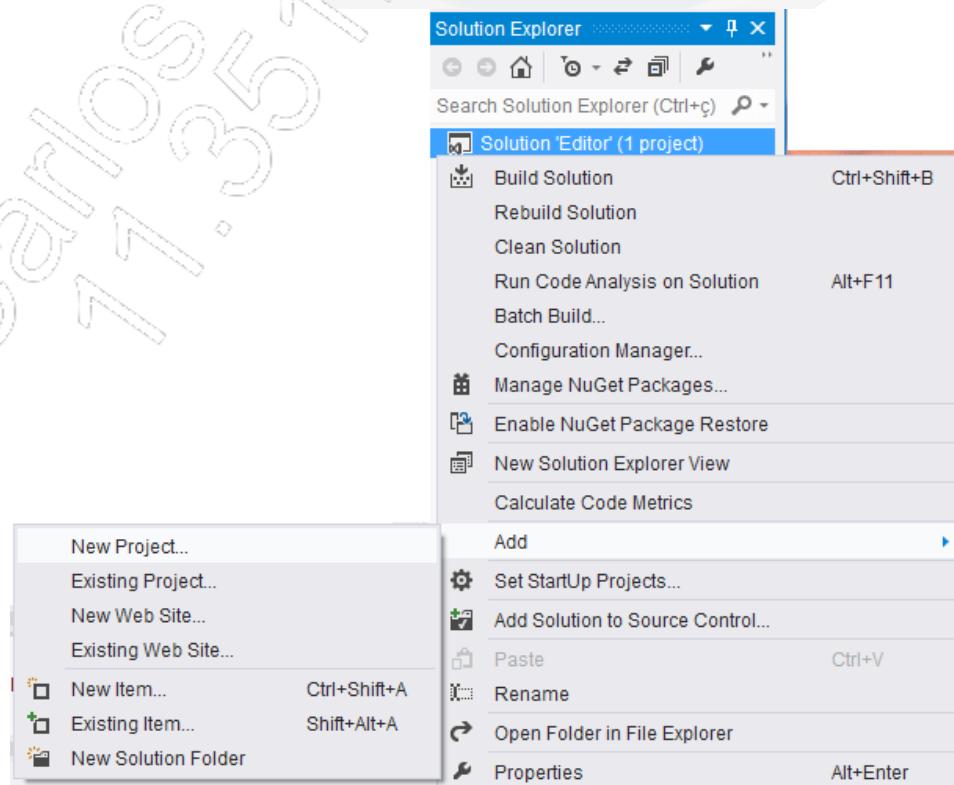
13.2. Instalação do InstallShield LE

Vejamos, a seguir, como instalar o **InstallShield Limited Edition**:

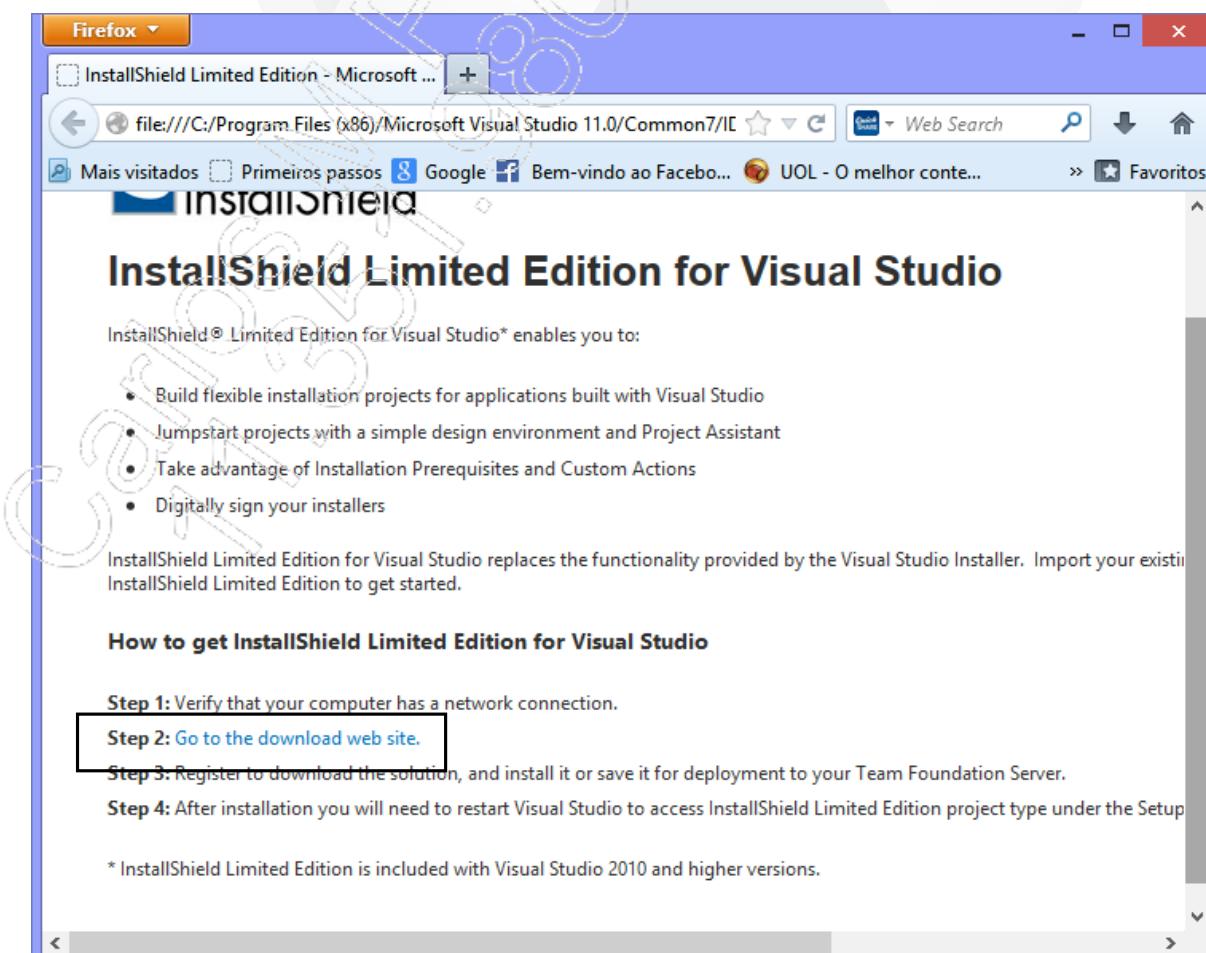
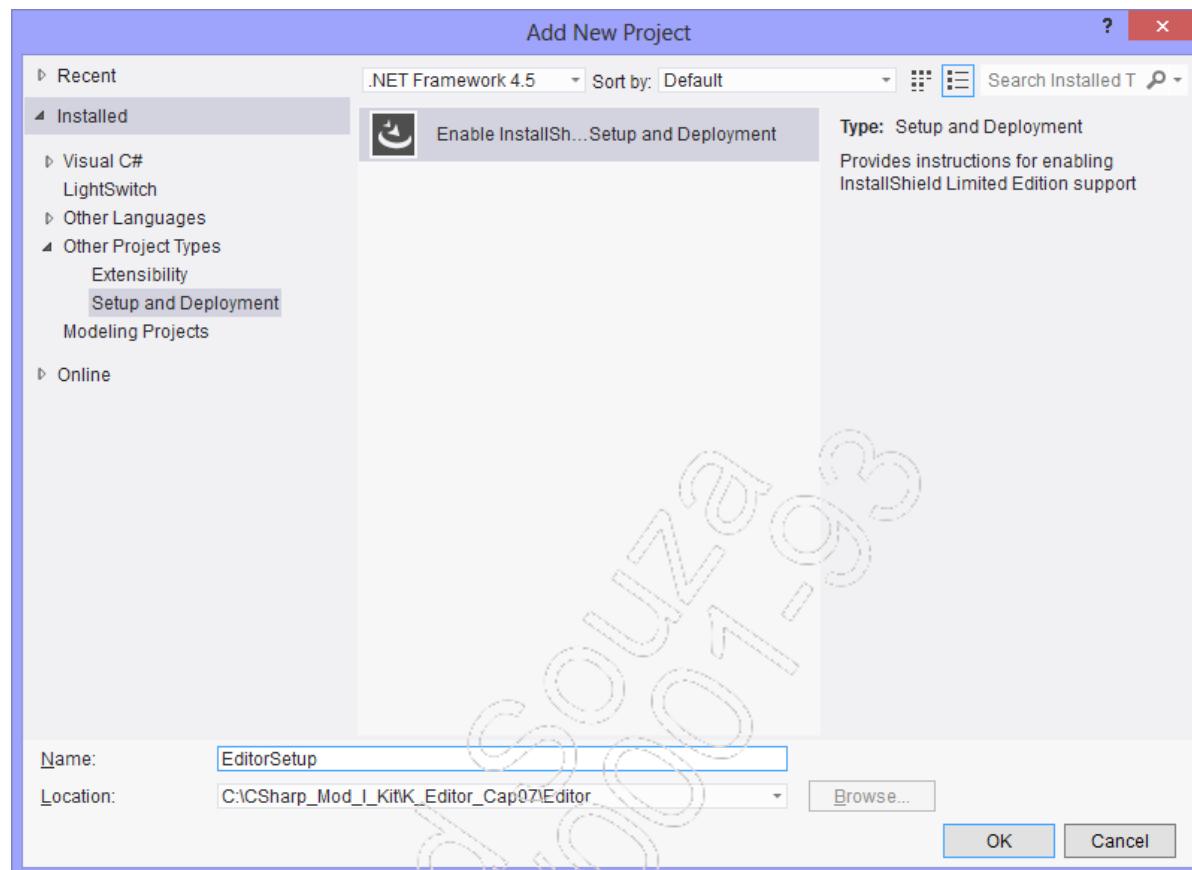
1. Abra o projeto **Editor** para o qual será criado um setup de instalação;



2. Na mesma solução, tente adicionar um novo projeto:

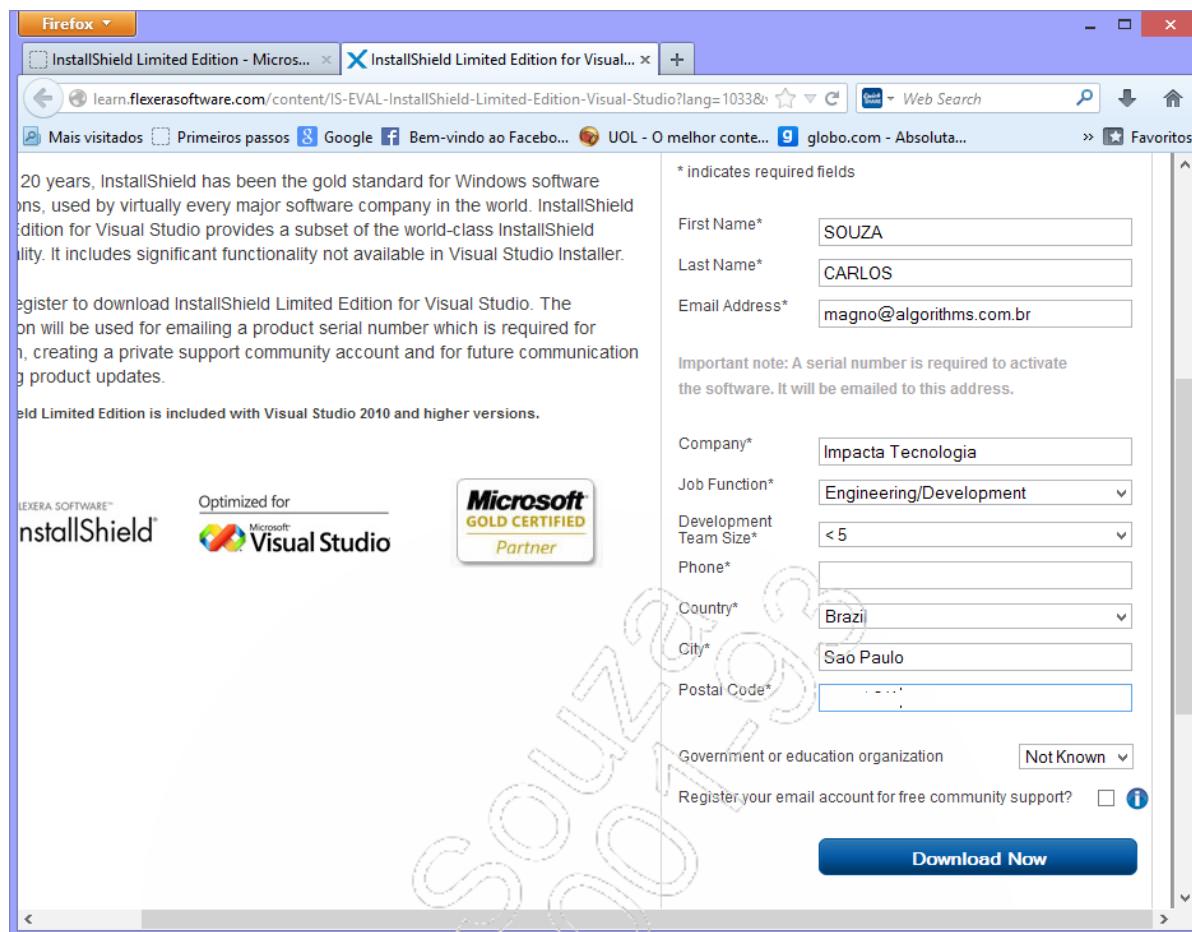


3. Na janela **Add New Project**, selecione **Installed / Other Project Types / Setup and Deployment / Enable InstallShield...**, nomeie o projeto e clique em **OK**. Será exibida a página para baixar o **InstallShield**:

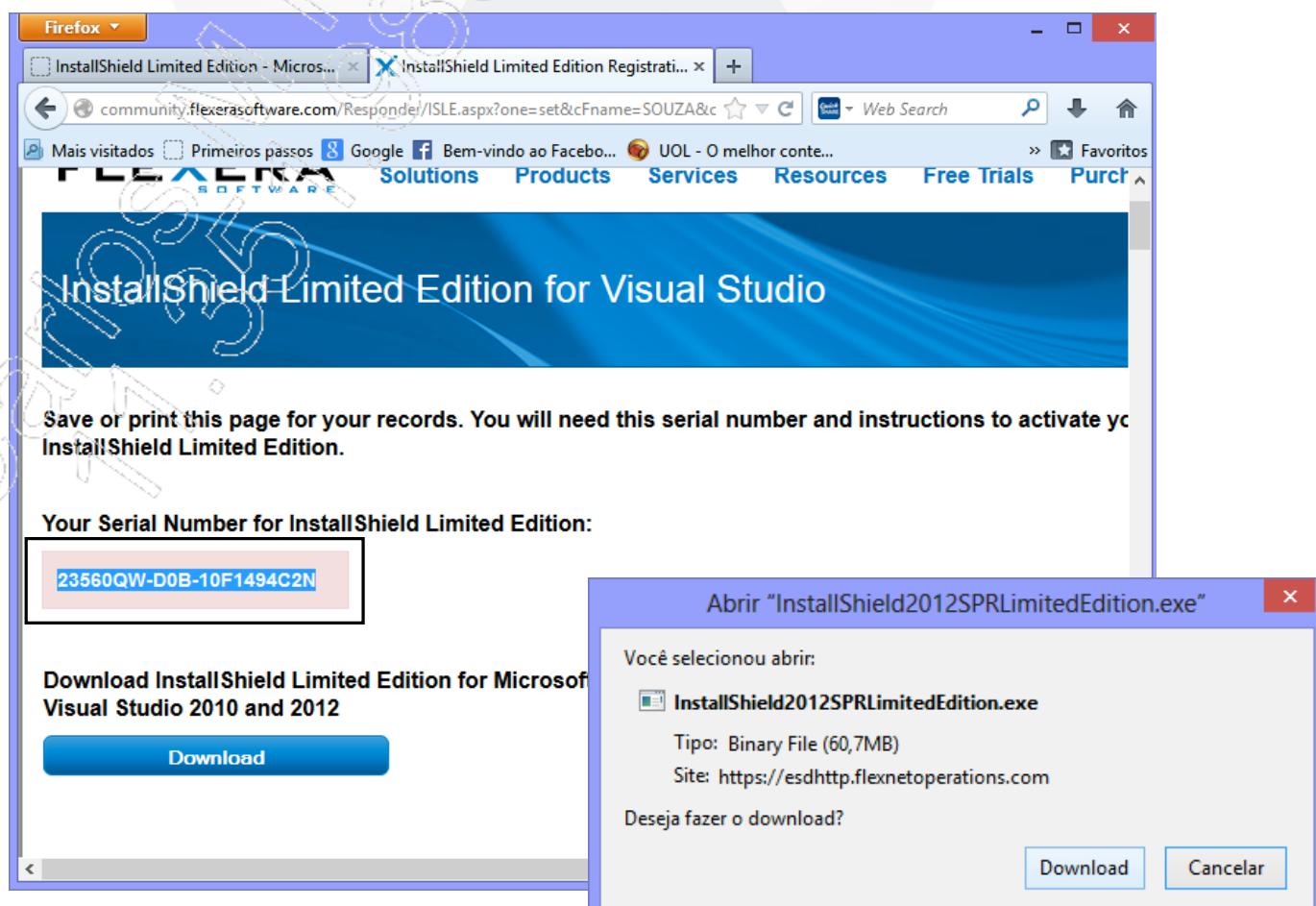


C# - Módulo I

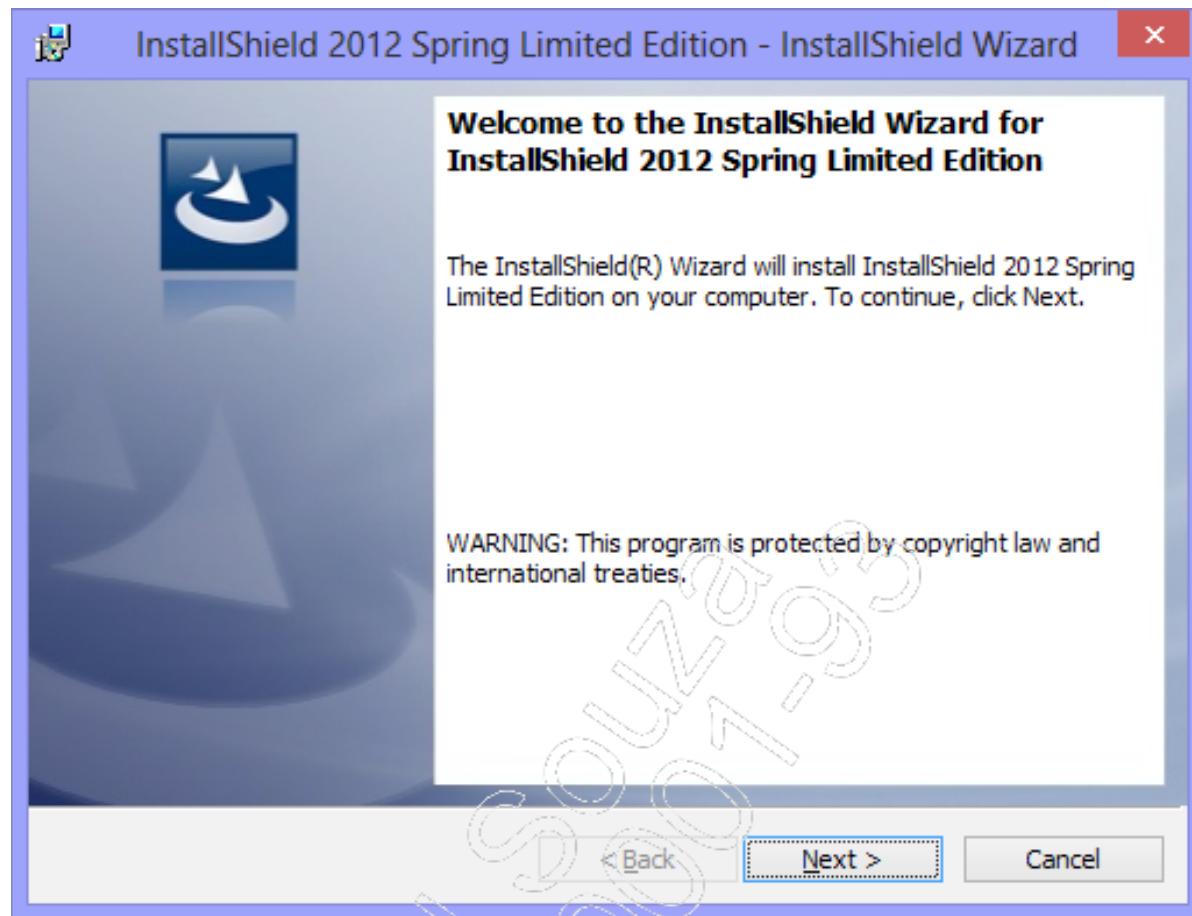
4. Insira os seus dados e clique em Download Now:



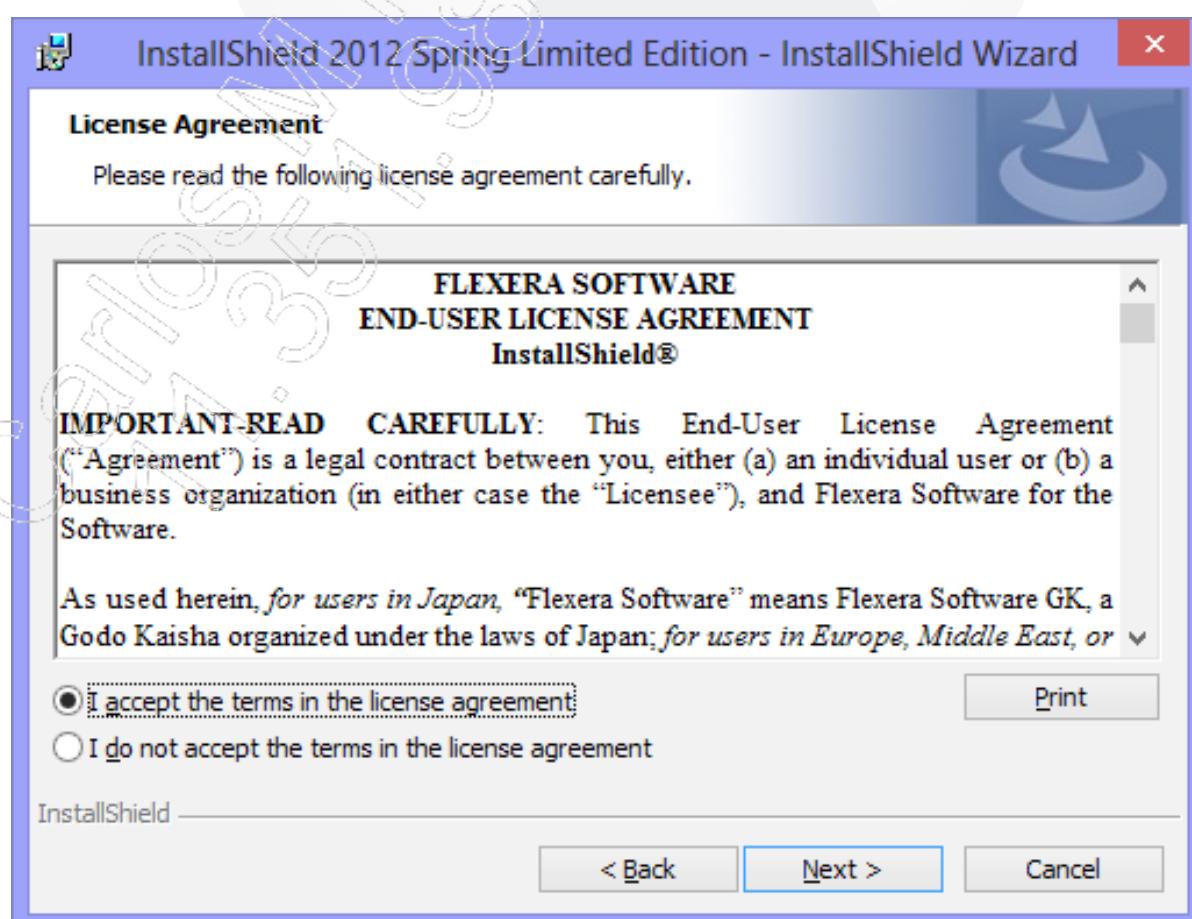
5. Anote o número de série, que será necessário mais adiante, e clique em Download nas duas telas a seguir:



6. Execute o programa baixado. Para isso, clique em **Next**:

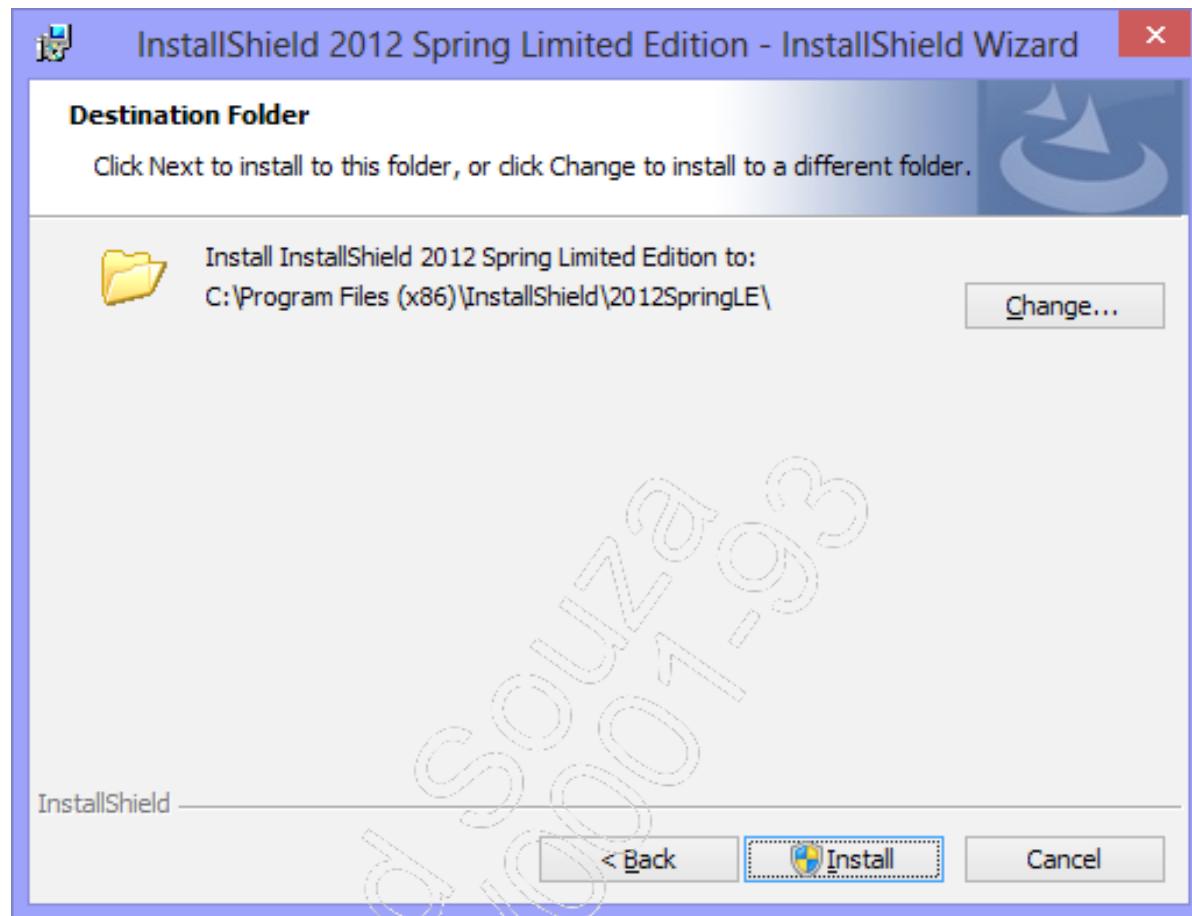


7. Aceite os termos de licença e clique em Next;



C# - Módulo I

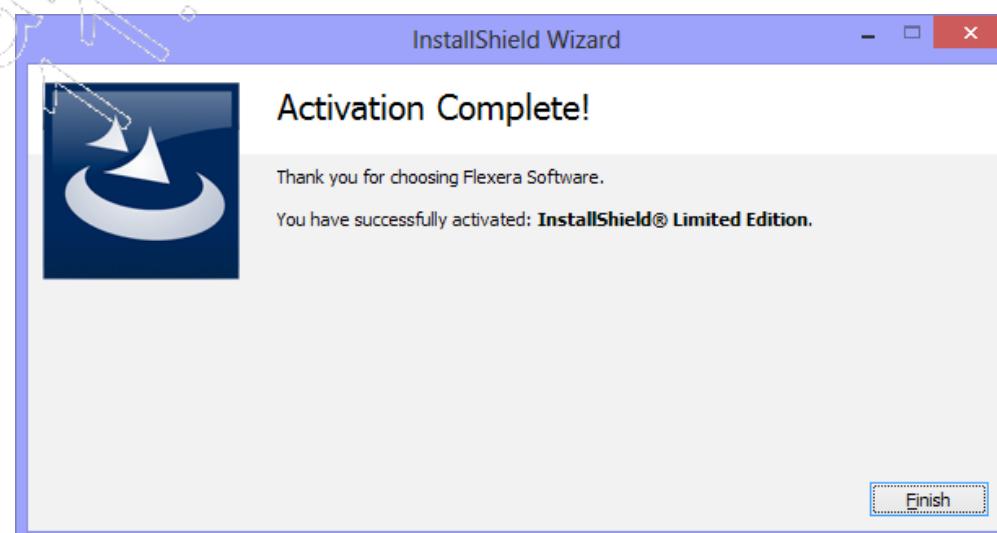
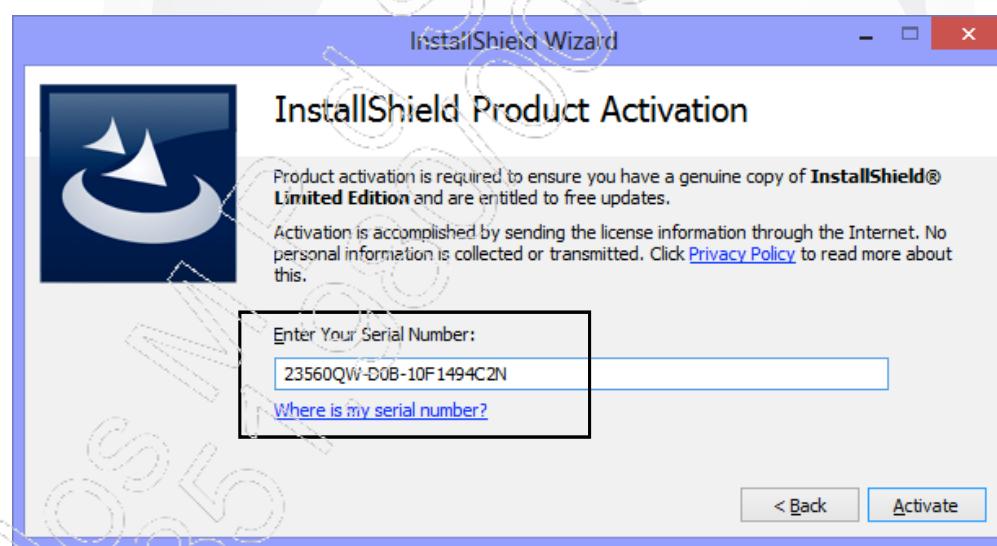
8. Acompanhe o processo até o final, clicando em **Install** e, em seguida, em **Finish**:



9. Nesta etapa, você deverá ativar o **InstallShield**. Para isso, selecione a opção indicada a seguir e clique em **Next**:



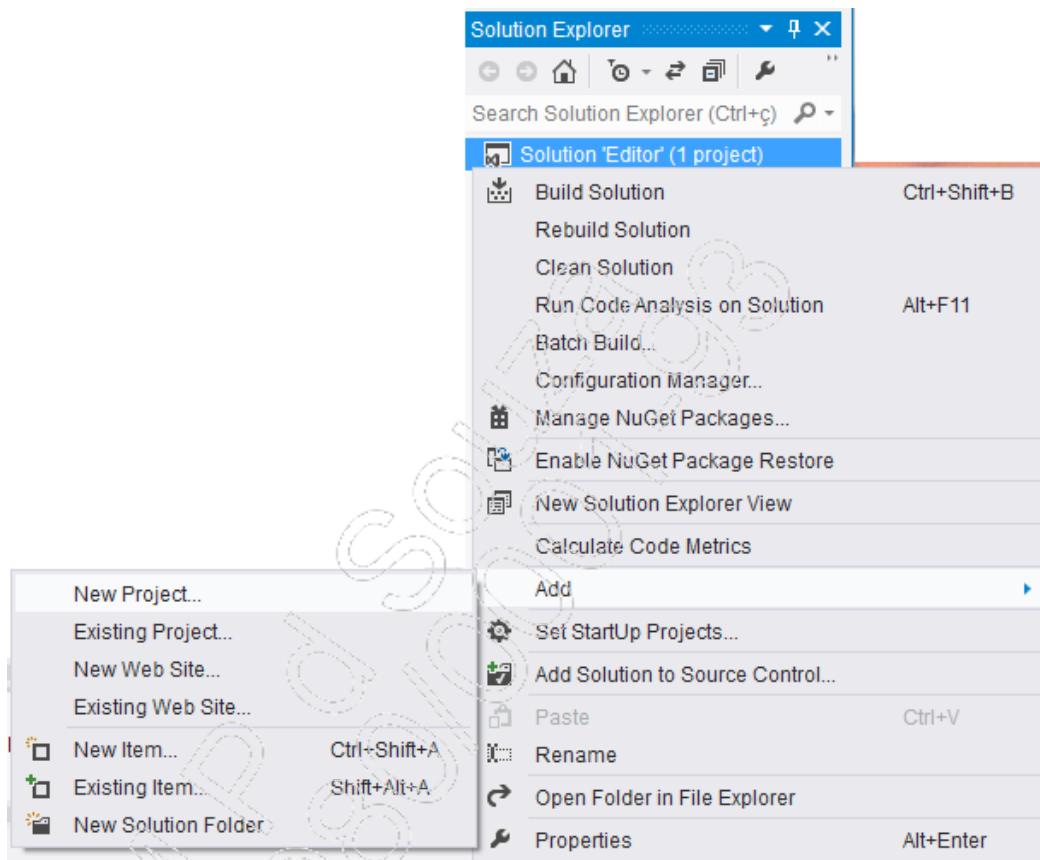
10. Informe o número de série anotado anteriormente, clique em **Activate** e, em seguida, em **Finish**.



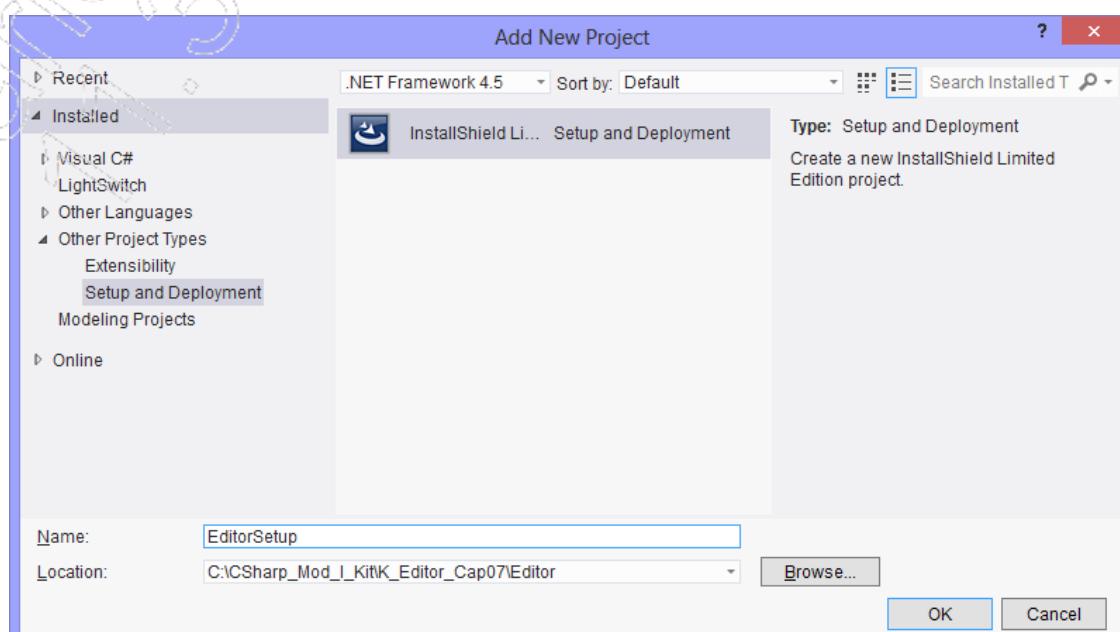
13.3. Usando o InstallShield

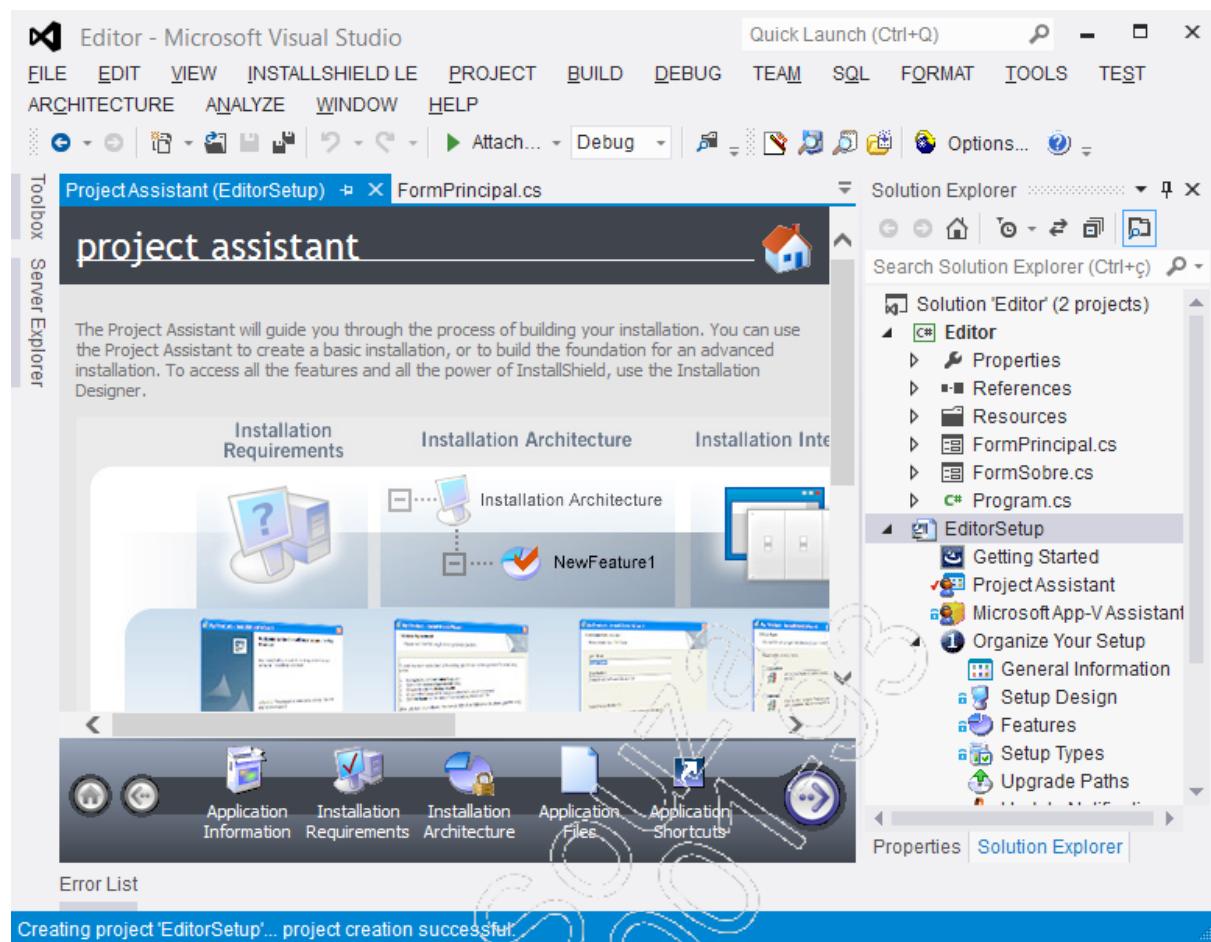
Vejamos agora como utilizar o **InstallShield**:

1. Feche o Visual Studio 2012 e abra-o novamente. Reabra o projeto **Editor** e repita o procedimento necessário para criar um projeto de setup;

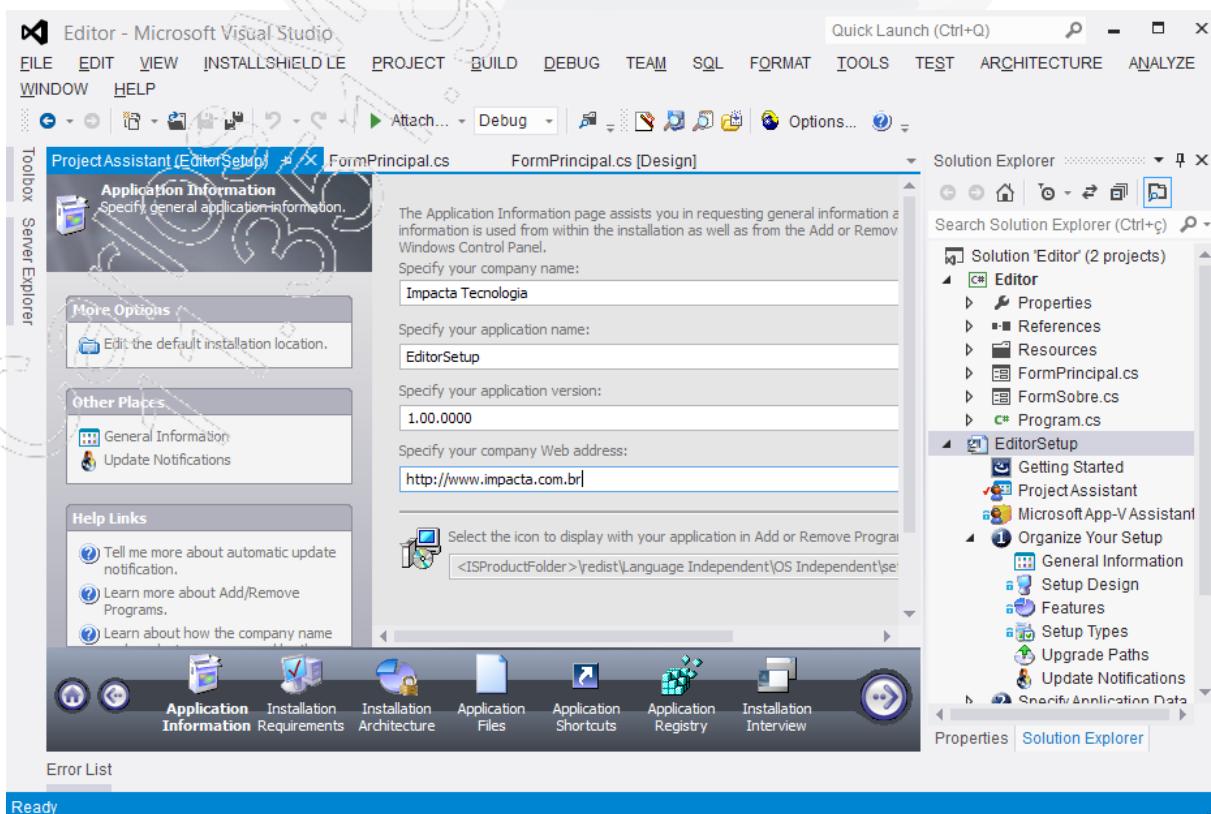


2. Na janela **Add New Project**, selecione **Installed / Other Project Types / Setup and Deployment / InstallShield...**, nomeie o projeto e clique em **OK**. Será exibida a página **Project Assistant**:



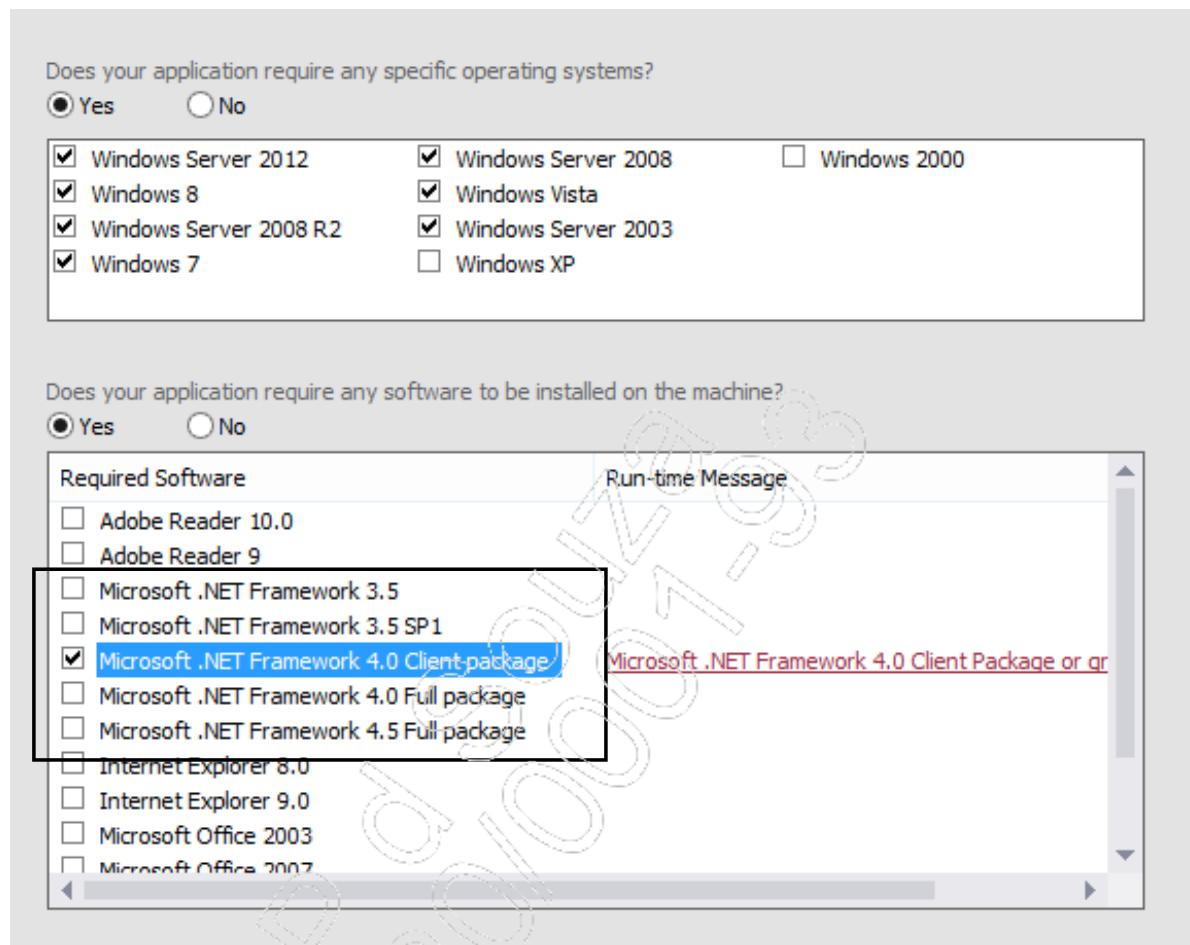


3. Clique em **Application Information** para informar o nome da empresa, versão, etc, de acordo com a imagem adiante:



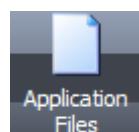


4. Clique em **Installation Requirements** para selecionar as opções em destaque a seguir:

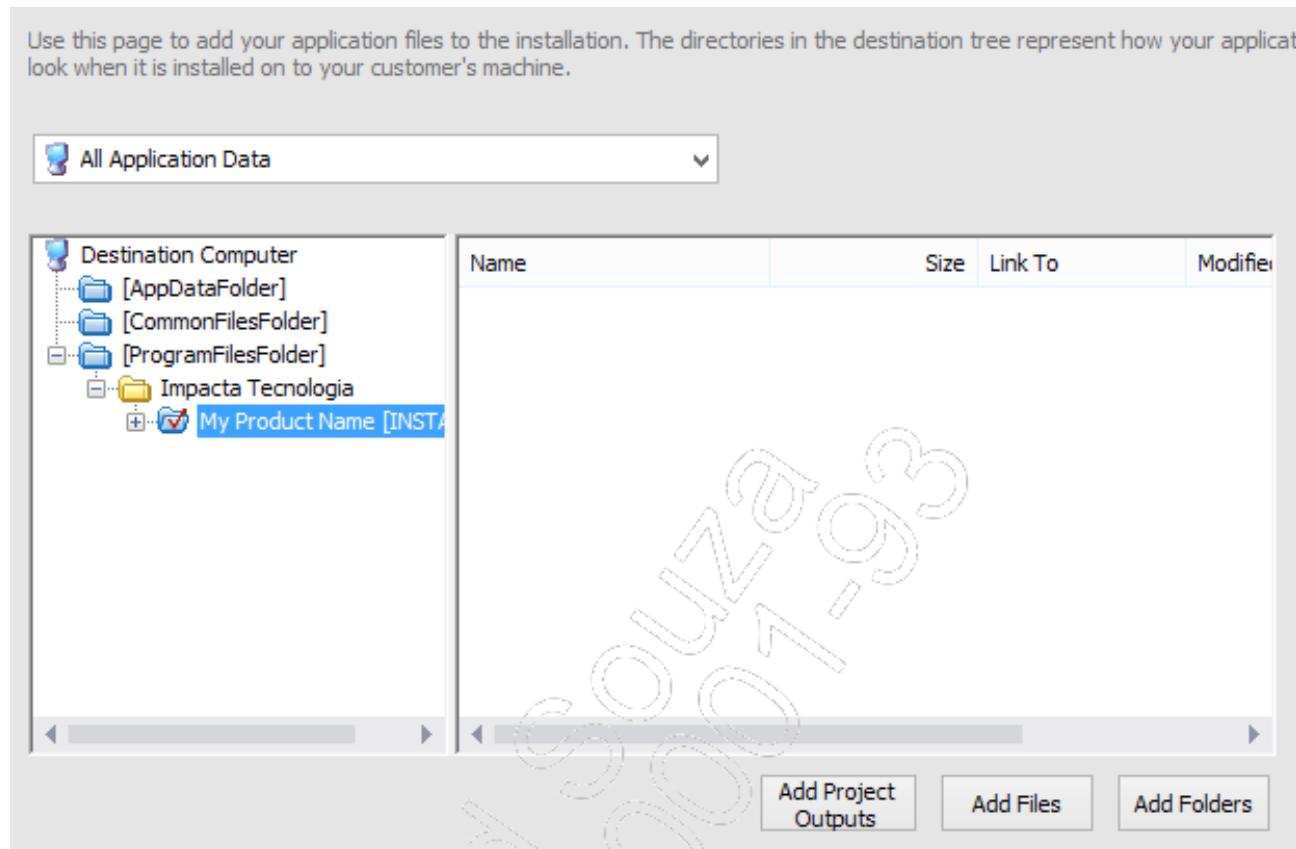


A opção **Installation Architecture** está disponível somente na versão full.

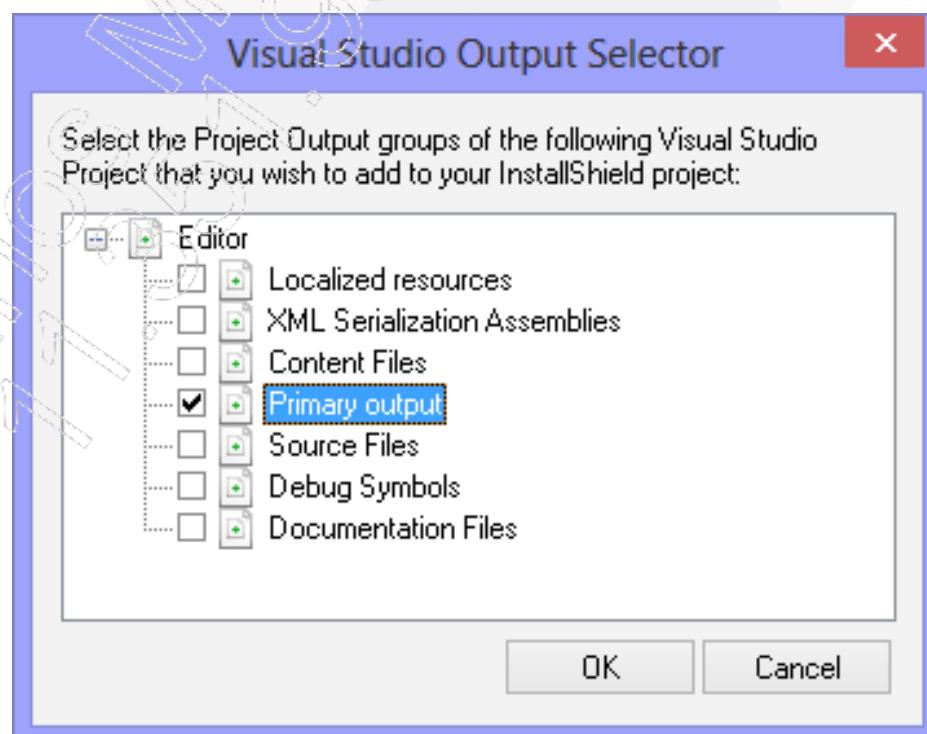




5. Em **Application Files**, você encontra os arquivos da aplicação. Clique no botão **Add Project Outputs**;

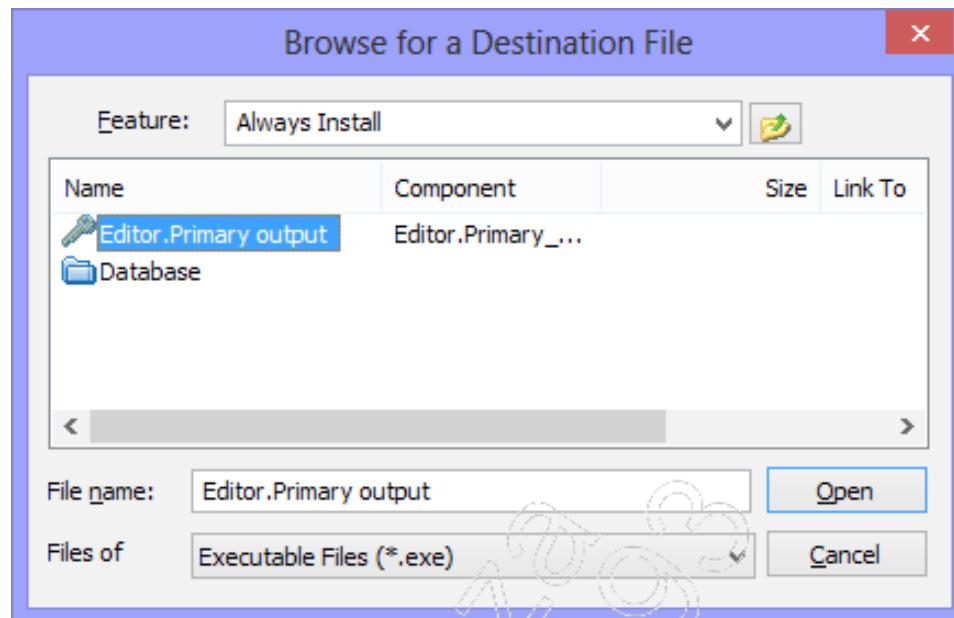


6. Selecione **Primary output**, que representa o executável da aplicação, e clique em **OK**;



C# - Módulo I

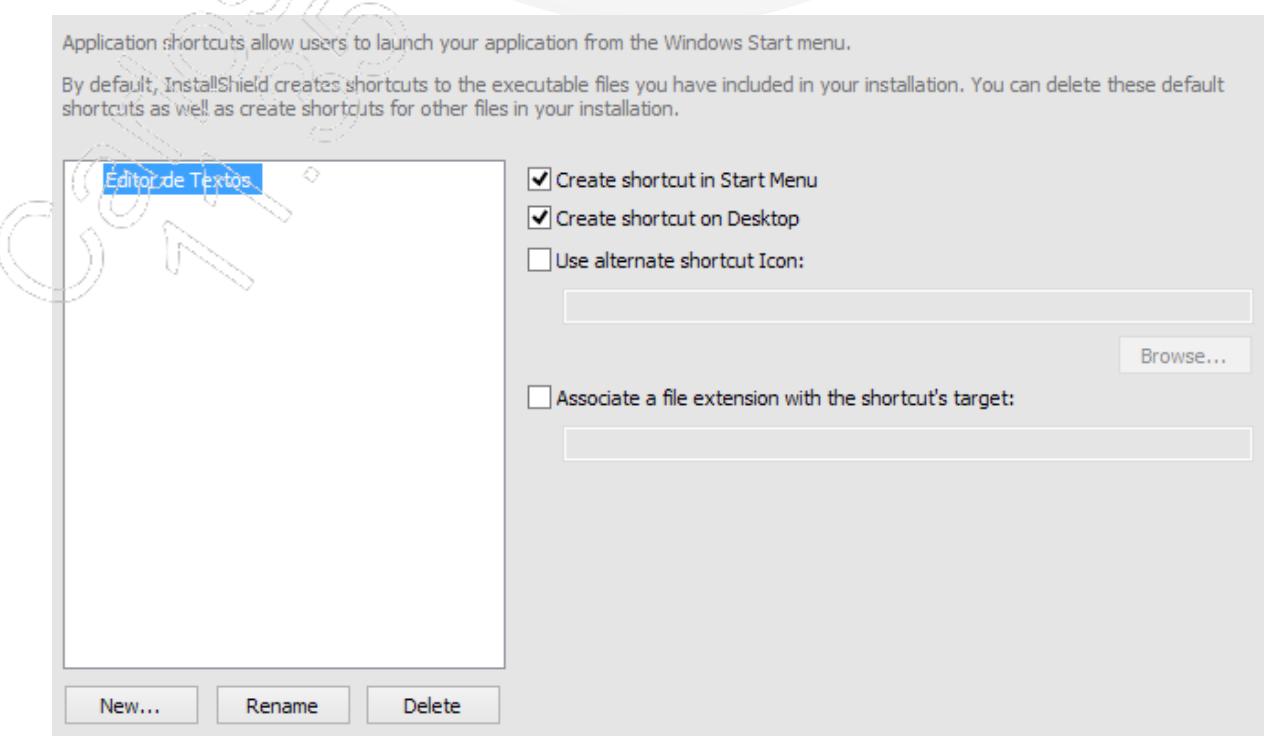
7. Se a aplicação necessitar de arquivos adicionais, como imagens, sons, entre outros, selecione **Add Files** para adicionar esses arquivos;



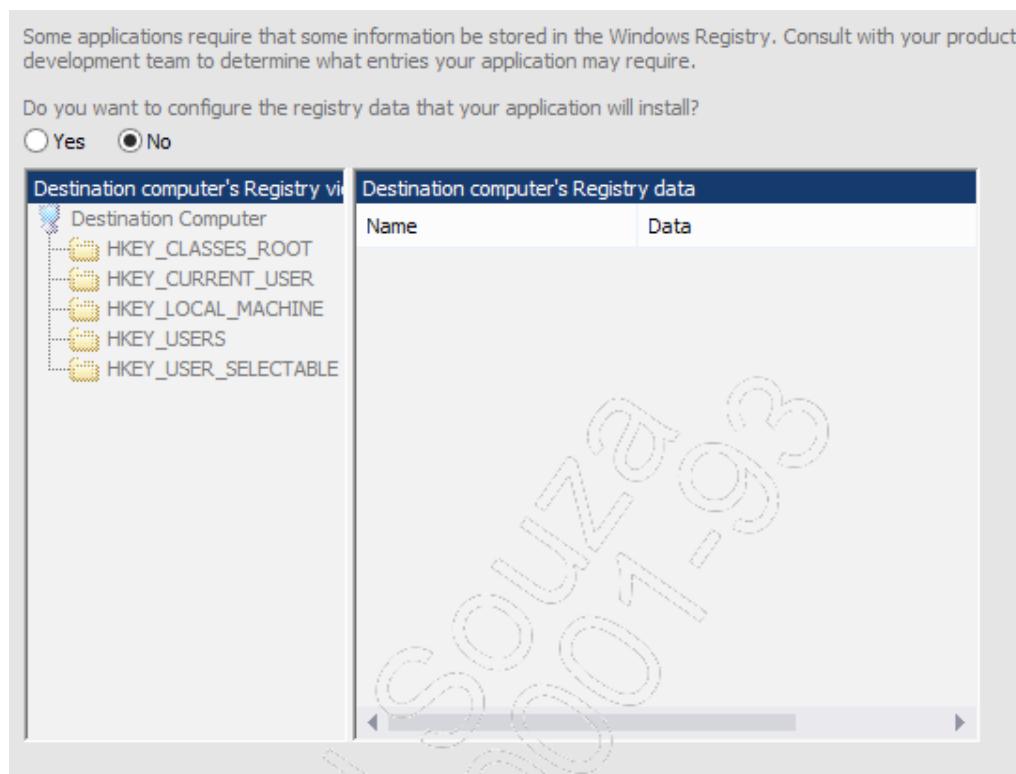
8. Em **Application Shortcuts**, você encontrará os atalhos para a aplicação. Clique no botão **New** e depois **Rename** para colocar um nome no atalho;

9. Ainda nessa opção, selecione **Create shortcut in Start Menu** para incluir esse atalho no menu **Iniciar**;

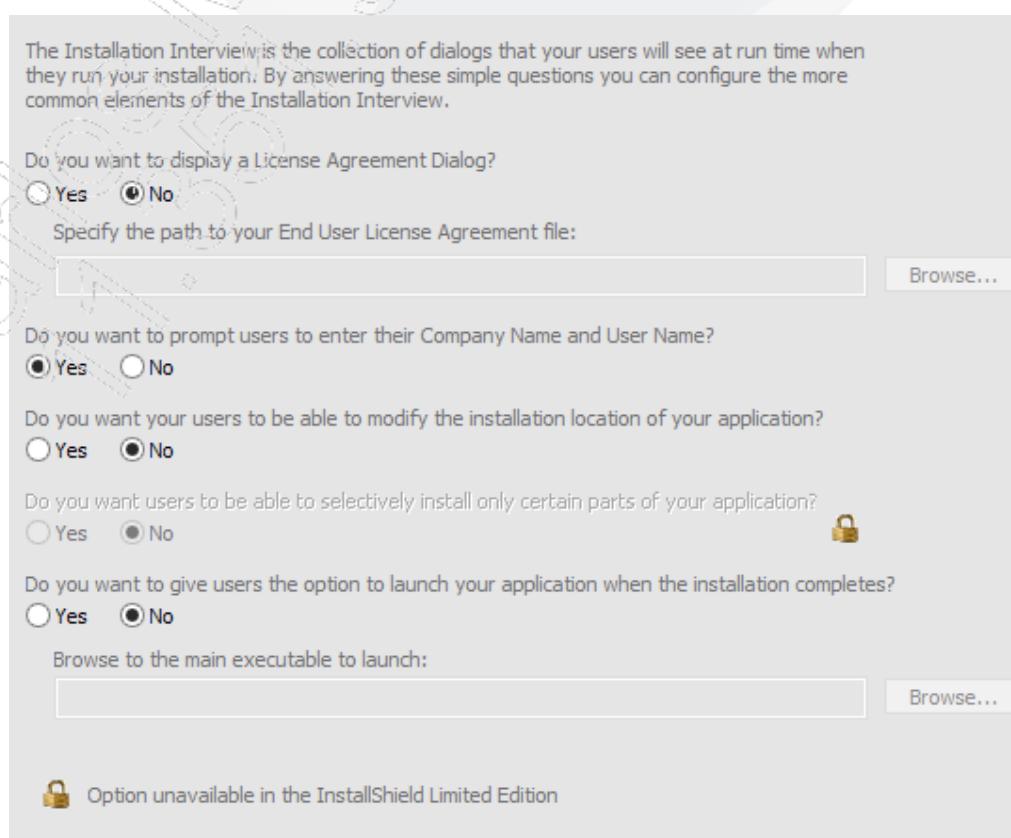
10. Selecione também **Create shortcut on Desktop** para criar um atalho no desktop;



11. Use a opção **Application Registry** para fazer alterações no registro do Windows. Neste caso, selecione **No**:

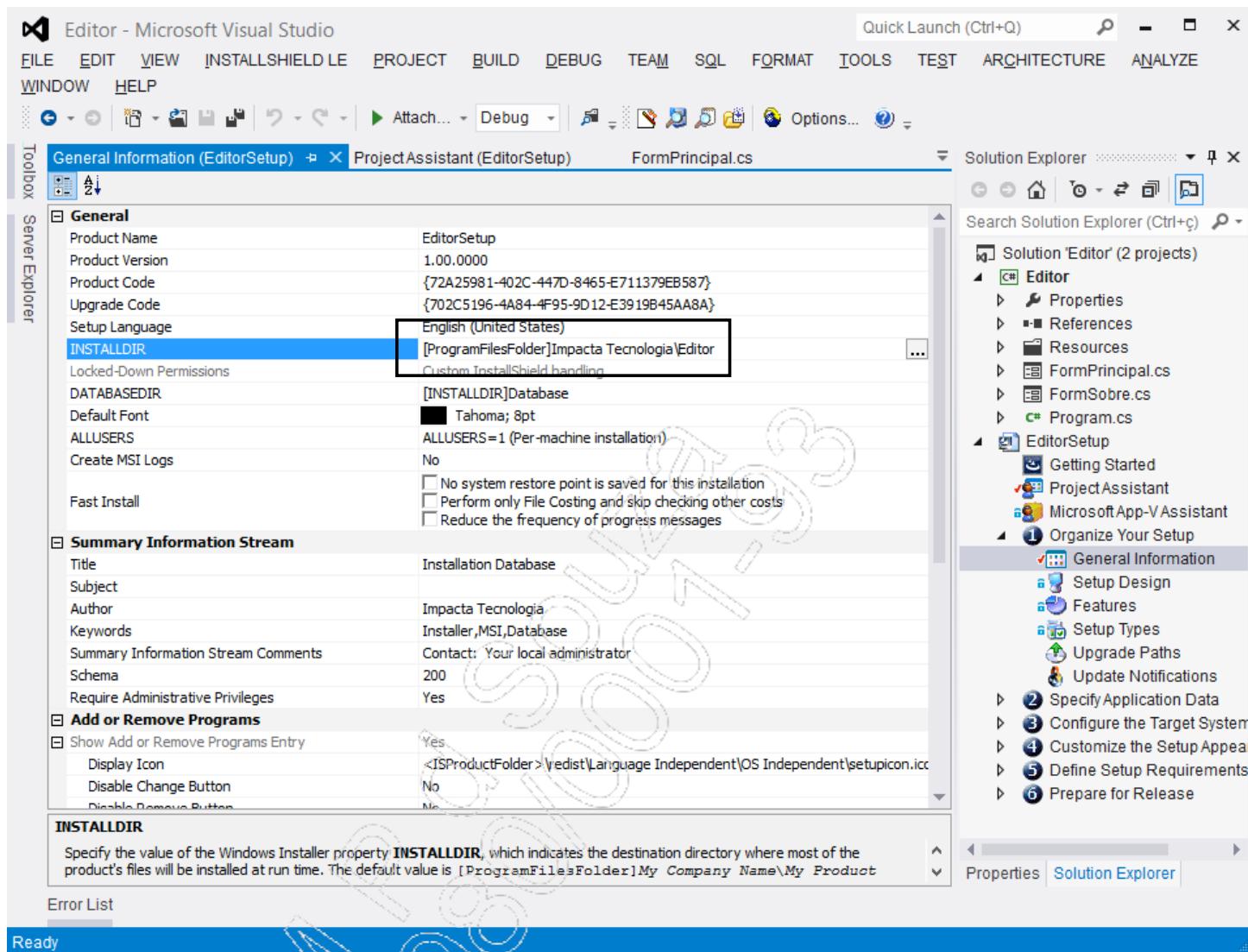


12. Em **Application Interview** são exibidas as telas do processo de instalação. Marque as seguintes opções desta tela:

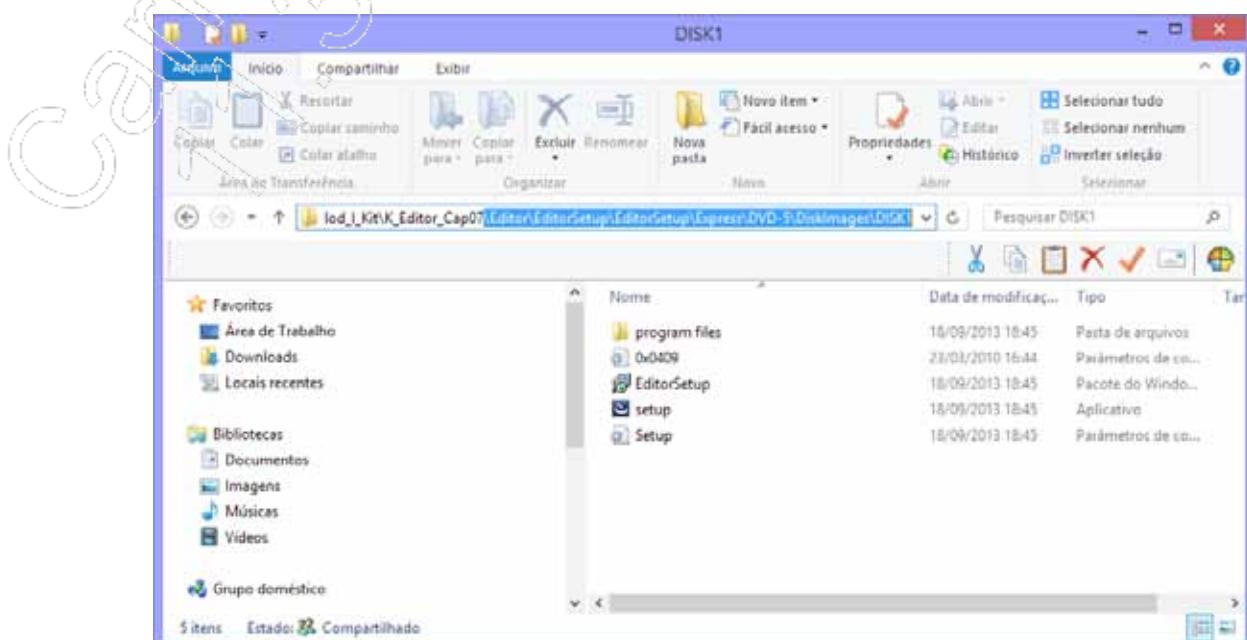


C# - Módulo I

13. Em **General Information**, está disponível o resumo das informações do setup. Altere o nome da pasta do projeto;



14. Compile o projeto de setup e o **InstallShield** criará uma pasta com o programa de instalação a partir do projeto inicial (...\\EditorSetup\\Express\\DVD-5\\DiskImages\\DISK1).



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Precisamos instalar o **InstallShield Limited Edition** no Visual Studio 2012, pois a opção **Setup Project** não existe mais;
- O **InstallShield** é um gerador de setup de instalação que existe há cerca de 10 anos. A versão **InstallShield for Visual Studio 2012 LE (Limited Edition)**, criada recentemente, pode ser baixada sem custos pela Internet.



