



Colégio  
*Pedro II*

# PROGRAMAÇÃO O.O. (C#)

---

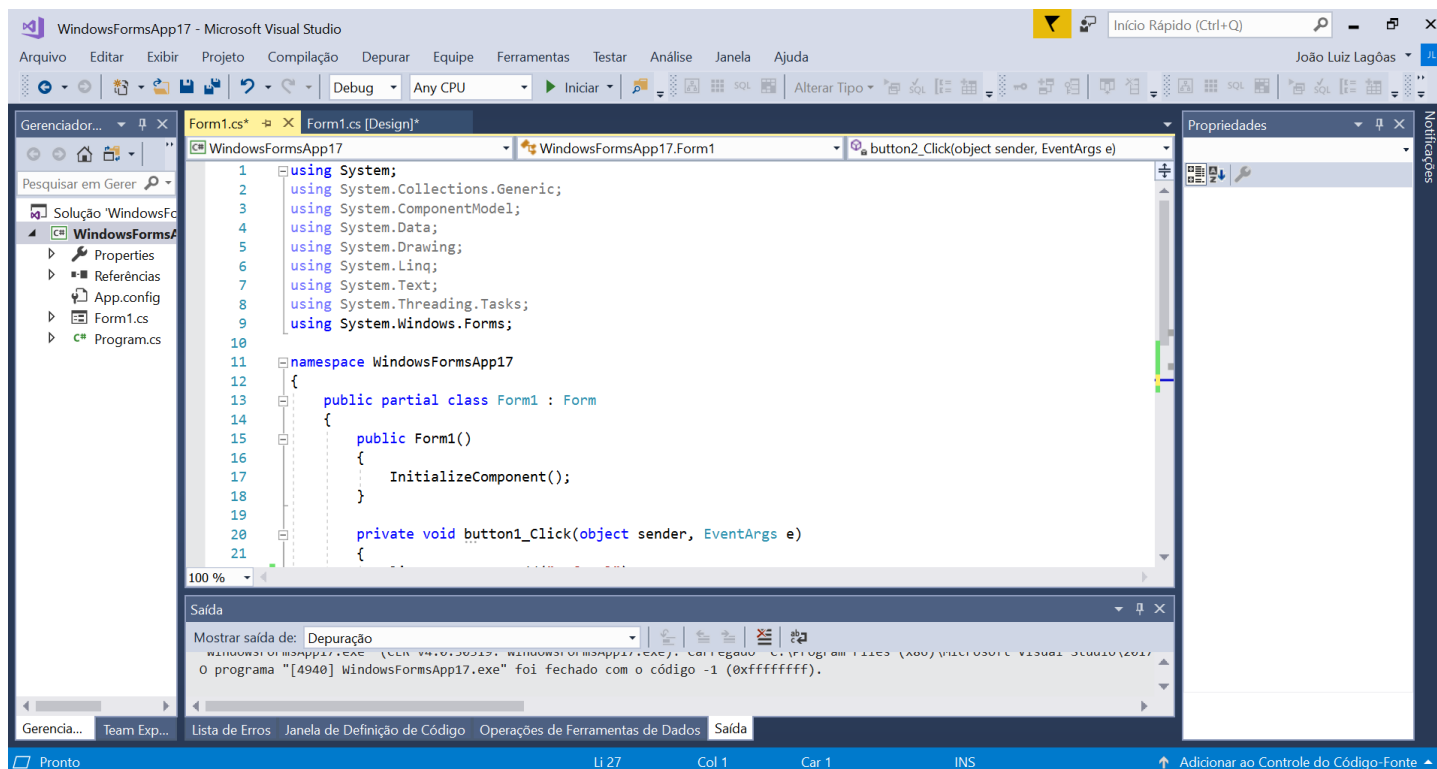


Propriedades, Eventos e mais Controles!  
**Professor:** João Luiz Lagôas



# Introdução




- Uma interface de usuário gráfica (GUI) permite que um usuário interaja visualmente com um programa.
- Como um exemplo de GUI, considere a própria interface do Visual Studio 2017:



# Introdução

## Controles

- GUIs são construídos através de **controles** (ou muitas vezes conhecidos como *widgets*).
- Os controles são essencialmente objetos de classes encontradas no *namespace Forms* e permitem que usuários interajam com a aplicação através do mouse, teclado ou outra forma de entrada. Alguns controles que já conhecemos até agora são listados abaixo:

	<b>Button</b>	Representa um controle de botão do Windows.
	<b>Label</b>	Representa um rótulo padrão do Windows.
	<b>TextBox</b>	Representa um controle de caixa de texto do Windows.

*namespace  
Forms*

# Introdução

## Controles

- Através da **Caixa de Ferramentas**, podemos construir nosso formulário (*Windows Form*) adicionando novos **controles**:



Controles mais usados →

Exibir		
	Gerenciador de Soluções	Ctrl+Alt+L
	Team Explorer	Ctrl+], Ctrl+M
	Gerenciador de Servidores	Ctrl+Alt+S
	Cloud Explorer	Ctrl+], Ctrl+X
	Pesquisador de Objetos do SQL Server	Ctrl+], Ctrl+S
	Janela de Indicadores	Ctrl+K, Ctrl+W
	Hierarquia de Chamada	Ctrl+Alt+K
	Modo de Exibição de Classe	Ctrl+Shift+C
	Janela de Definição de Código	Ctrl+], D
	Pesquisador de Objetos	Ctrl+Alt+J
	Lista de Erros	Ctrl+], E
	Saída	Ctrl+Alt+O
	Lista de Tarefas	Ctrl+], T
	<b>Caixa de Ferramentas</b>	<b>Ctrl+Alt+X</b>
	Notificações	Ctrl+W, N
	Localizar Resultados	▶
	Outras Janelas	▶
	Barra de Ferramentas	▶
	Tela Inteira	Shift+Alt+Enter
	Todas as Janelas	Shift+Alt+M
	Navegar para Trás	Ctrl+-
	Navegar para Frente	Ctrl+Shift+-
	Próxima Tarefa	
	Tarefa Anterior	
	Janela de Propriedades	F4
	Páginas de Propriedades	Shift+F4

### ▲ Controles Comuns

	Ponteiro
	Button
	CheckBox
	CheckedListBox
	ComboBox
	DateTimePicker
	Label
	LinkLabel
	ListBox
	ListView
	MaskedTextBox
	MonthCalendar
	NotifyIcon
	NumericUpDown
	PictureBox
	ProgressBar
	RadioButton
	RichTextBox
	TextBox
	ToolTip
	TreeView
	WebBrowser

# Introdução

## Forms

- Um *Form* é um “container” para controles. Quando um item é arrastado da Caixa de Ferramentas para o Form, o Visual Studio gera código que cria o objeto e ajusta os seus atributos/propriedades.
- Essencialmente, ao se criar um projeto de *Windows Form*, você terá acesso às funcionalidades de Designer e a 3 arquivos.

- **Form1.cs**

- **Form1.Designer.cs**

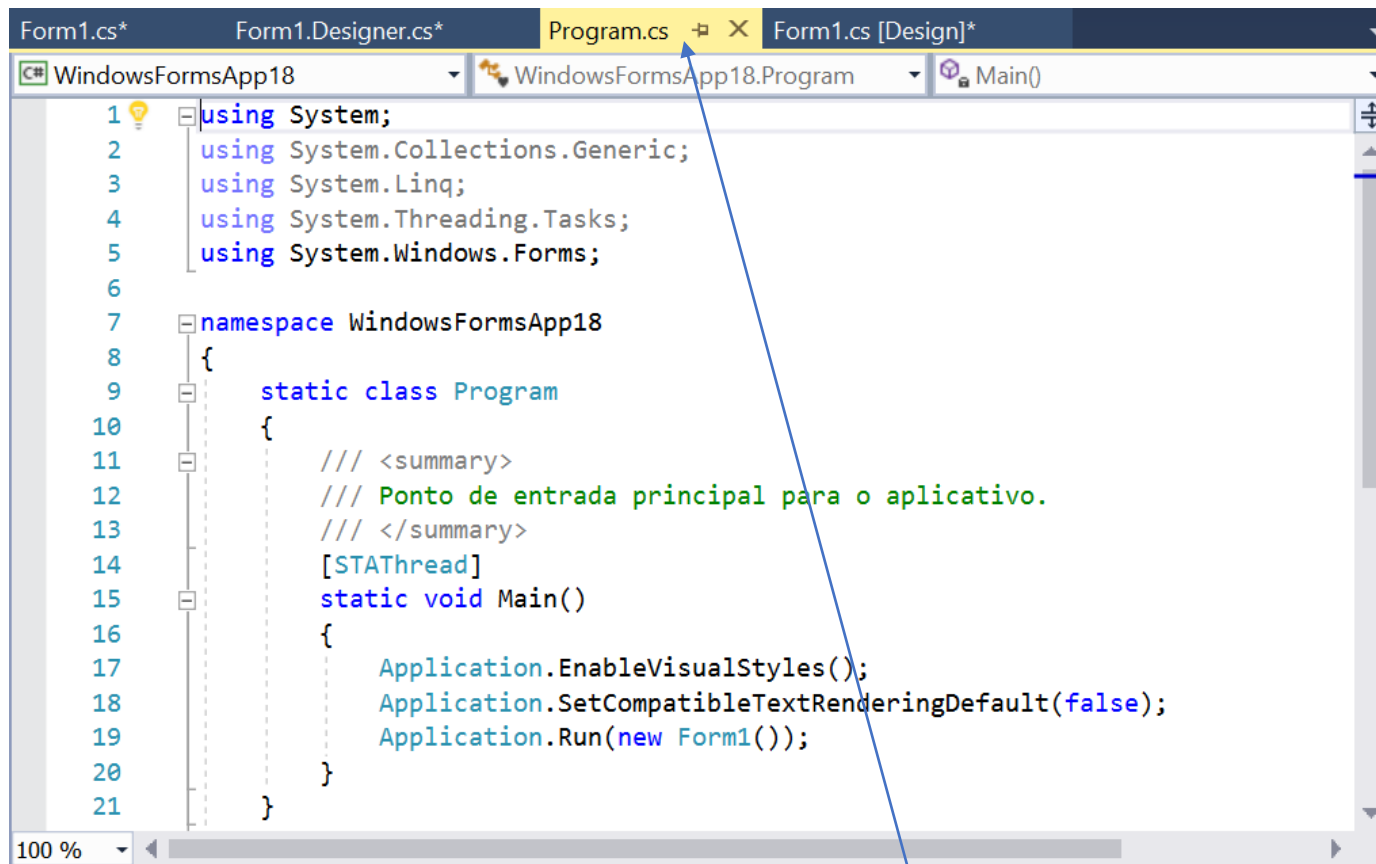
- **Program.cs**



**Classes parciais**

# Introdução

## Forms: Program.cs

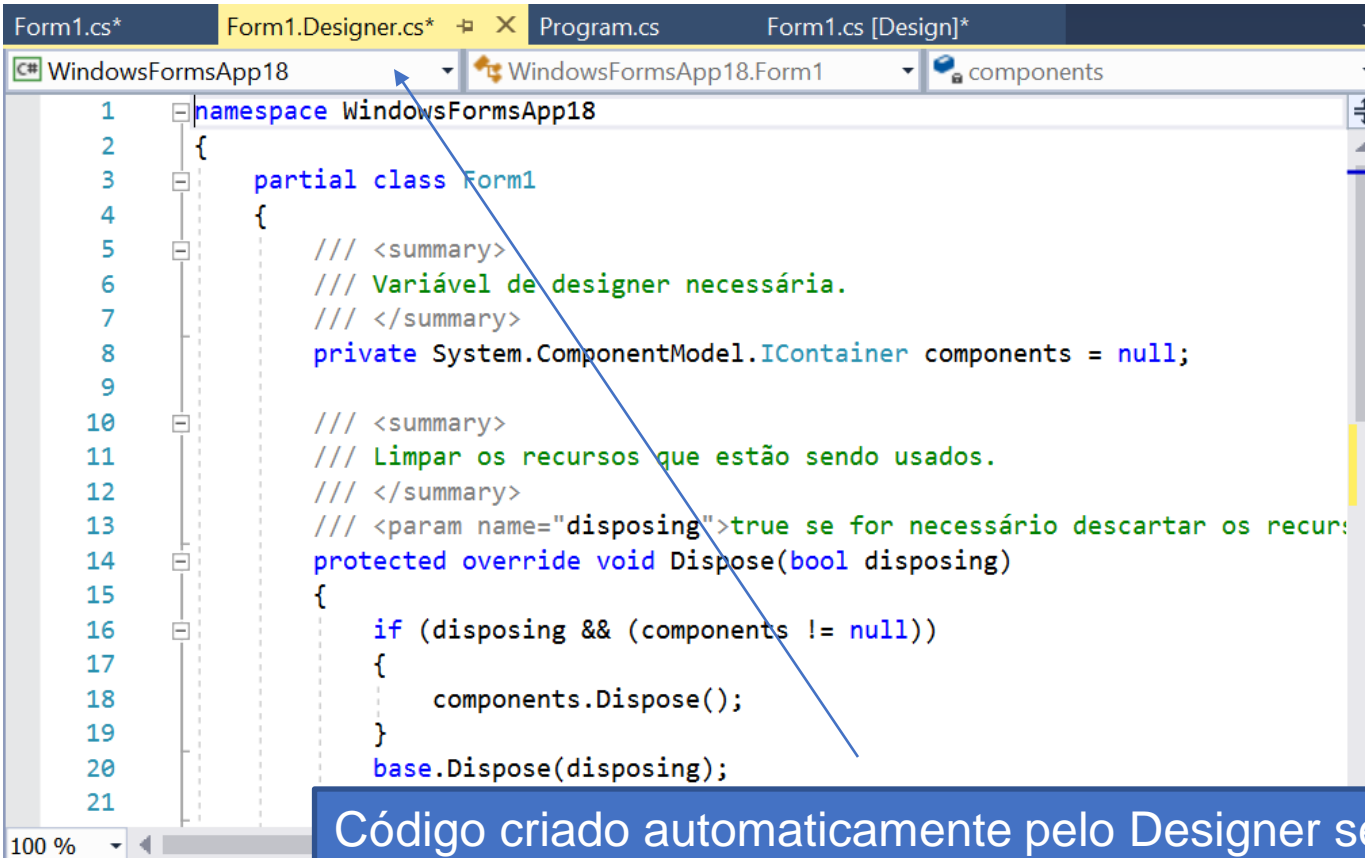


```
Form1.cs*   Form1.Designer.cs*   Program.cs   Form1.cs [Design]*
C# WindowsFormsApp18
WindowsFormsApp18.Program   Main()
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using System.Windows.Forms;
6
7  namespace WindowsFormsApp18
8  {
9      static class Program
10     {
11         /// <summary>
12         /// Ponto de entrada principal para o aplicativo.
13         /// </summary>
14         [STAThread]
15         static void Main()
16         {
17             Application.EnableVisualStyles();
18             Application.SetCompatibleTextRenderingDefault(false);
19             Application.Run(new Form1());
20         }
21     }
22 }
```

Fluxo inicial de código onde se encontra o método Main()

# Introdução

## Forms: Form1.Designer.cs

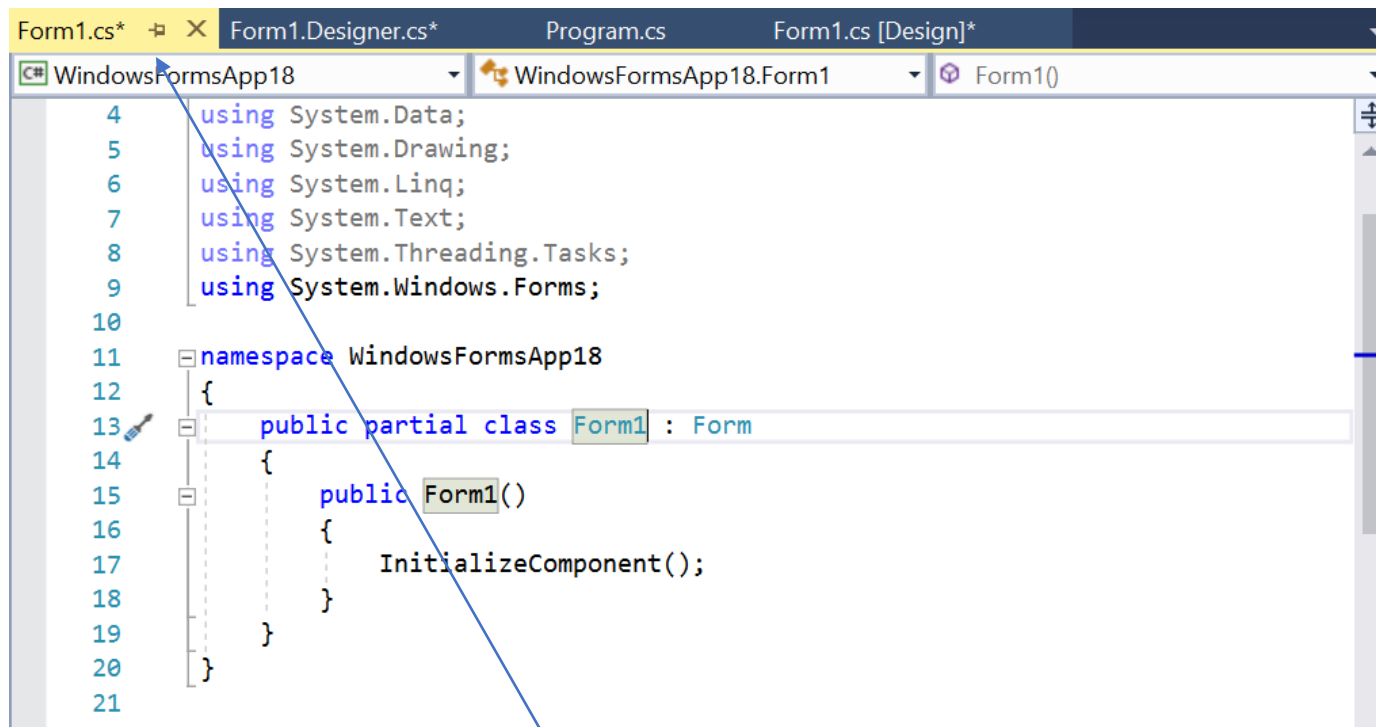


```
1 namespace WindowsFormsApp18
2 {
3     partial class Form1
4     {
5         /// <summary>
6         /// Variável de designer necessária.
7         /// </summary>
8         private System.ComponentModel.IContainer components = null;
9
10        /// <summary>
11        /// Limpar os recursos que estão sendo usados.
12        /// </summary>
13        /// <param name="disposing">true se for necessário descartar os recursos gerenciados.
14        protected override void Dispose(bool disposing)
15        {
16            if (disposing && (components != null))
17            {
18                components.Dispose();
19            }
20            base.Dispose(disposing);
21        }
22    }
23 }
```

Código criado automaticamente pelo Designer sempre que alguma mudança é feita no formulário através da interface gráfica.

# Introdução

## Forms: Form1.cs



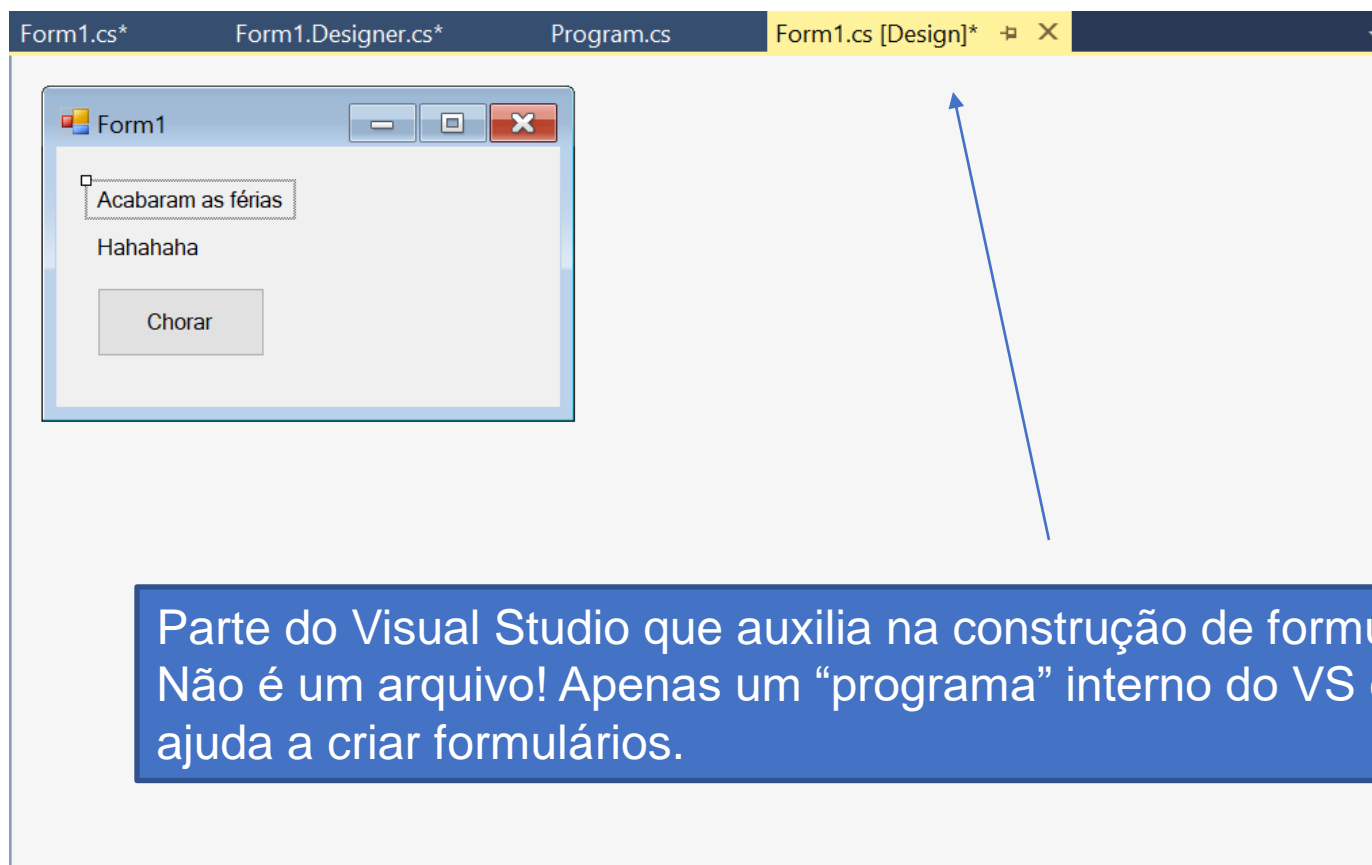
```
Form1.cs*  Form1.Designer.cs*  Program.cs  Form1.cs [Design]*
C# WindowsFormsApp18  WindowsFormsApp18.Form1  Form1()
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10
11 namespace WindowsFormsApp18
12 {
13     public partial class Form1 : Form
14     {
15         public Form1()
16         {
17             InitializeComponent();
18         }
19     }
20 }
21
```

Código criado pelo Visual Studio que representa o formulário mas que NÃO é alterado pelo Designer! Apenas o programador utiliza esta parte de código.



# Introdução

## Forms: funcionalidades de Designer



# Introdução

## Classes Parciais

- A IDE mantém o código gerado em um arquivo separado usando o conceito de **classes parciais** – classes que são divididas em múltiplos arquivos e se juntam em uma única classe pelo compilador.



**Form1.cs**



O programador implementa funcionalidades aqui!



**Form1.Designer.cs**

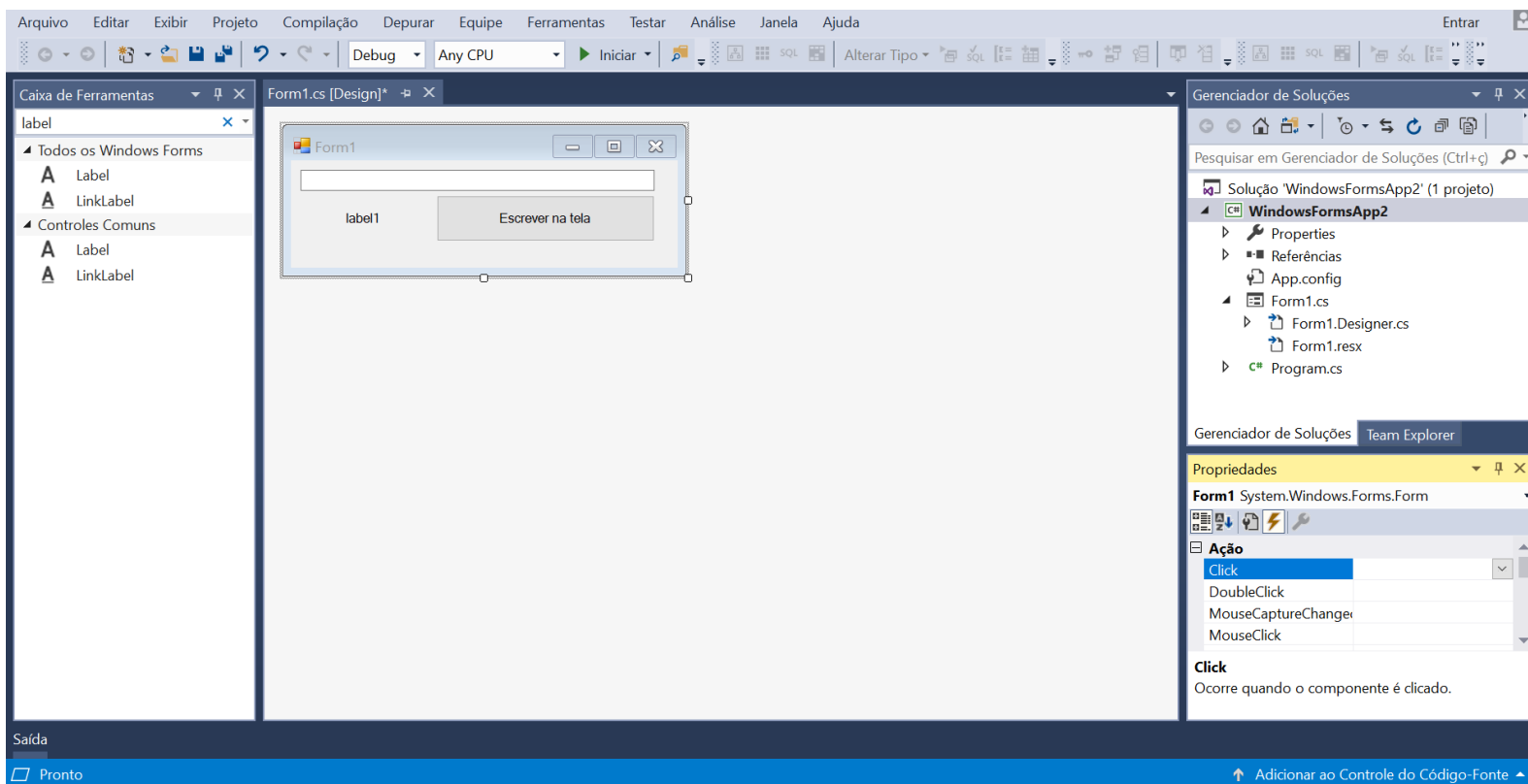


O Designer implementa funcionalidades aqui!  
Normalmente o programador não modifica esse código.

# Introdução

## Mecanismo de funcionamento do Designer

- Após criar um projeto de Windows Form, é comum se **adicionar controles** no formulário, **ajustar suas propriedades** e **implementar métodos** que tratam de **eventos** gerados pelos controles.



# Introdução

## Mecanismo de funcionamento do Designer

- Após criar um projeto de Windows Form, é comum se **adicionar controles** no formulário, **ajustar suas propriedades** e **implementar métodos** que tratam de **eventos** gerados pelos controles.

The image shows a screenshot of the Visual Studio Windows Forms Designer interface. The interface is divided into several panes: a menu bar at the top, a toolbar, a 'Caixa de Ferramentas' (Toolbox) on the left, a central design area, a 'Gerenciador de Soluções' (Solution Explorer) on the right, and a 'Propriedades' (Properties) pane at the bottom right. The 'Caixa de Ferramentas' shows a list of controls under 'Controles Comuns', including 'Label' and 'LinkLabel'. The 'Gerenciador de Soluções' shows the project structure, including 'Form1.cs', 'Form1.Designer.cs', 'Form1.resx', and 'Program.cs'. The 'Propriedades' pane shows the properties of the selected 'Form1' component, including the 'Ação' (Action) section with 'Click', 'DoubleClick', 'MouseCaptureChange', and 'MouseDown' events. A text box in the center of the design area states: 'Ao se clicar em um evento, um método é criado em Form1.cs para que o programador implemente sua funcionalidade'. Three callout boxes are present: 'Controles' (red text) pointing to the Toolbox, 'Propriedades' (blue text) pointing to the Properties pane, and 'Eventos' (purple text) pointing to the 'Ação' section in the Properties pane.

Ao se clicar em um evento, um **método** é criado em Form1.cs para que o programador implemente sua funcionalidade

**Controles**

**Propriedades**

**Eventos**

# Propriedades



Colégio  
Pedro II

Revisão e mais detalhes



Encapsulamento



Herança



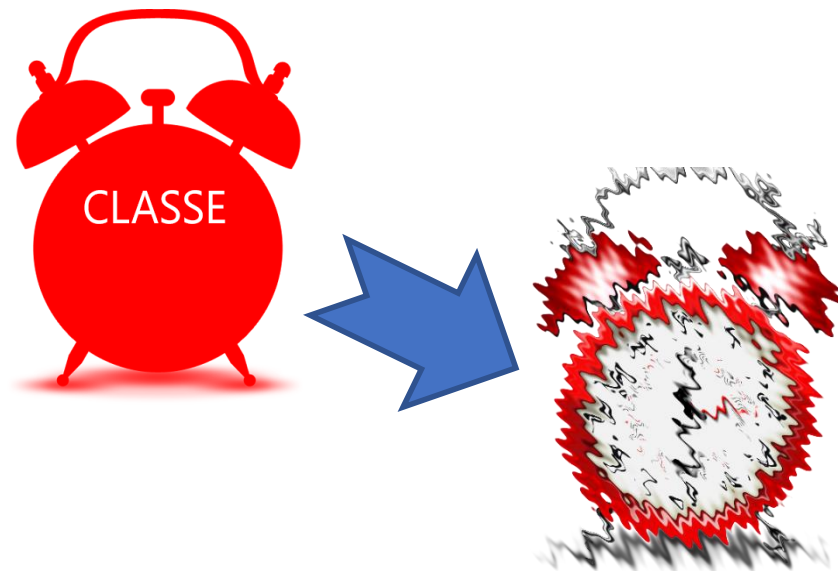
Polimorfismo

# Encapsulamento

## Revisão



- Quando você torna os seus atributos e métodos **públicos**, qualquer outra classe pode acessá-los.
- Tudo que sua classe faz e sabe torna-se um livro aberto para todas as outras classes em seu programa...
- Isso pode permitir que objetos apresentem estados inconsistentes e que o aplicativo comporte-se de formas que você nunca imaginou.



# Encapsulamento

## Exemplo Fazendeiro



Colégio  
**Pedro II**

```
class Fazendeiro
{
    public int numeroDeOvelhas = 10;
    public int sacosDeRacao;
    public const int multiplicadorDeRacao = 30;
}
```



```
class Lobo
{
    public bool ComerOvelha(Fazendeiro vitima)
    {
        if (vitima.numeroDeOvelhas >= 1)
        {
            vitima.numeroDeOvelhas--;
            return true;
        }
        else
            return false;
    }
}
```



# Encapsulamento

## Exemplo Fazendeiro



```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    Lobo lobo = new Lobo();

    lobo.ComerOvelha(pedro);
}
```



lobo.ComerOvelha(pedro)





# Encapsulamento

## Exemplo Fazendeiro



Colégio  
**Pedro II**

```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    Lobo lobo = new Lobo();

    lobo.ComerOvelha(pedro);
}
```

```
public bool ComerOvelha(Fazendeiro vitima)
{
    if (vitima.numeroDeOvelhas >= 1)
    {
        vitima.numeroDeOvelhas--;
        return true;
    }
    else
        return false;
}
```



lobo.ComerOvelha(pedro)



# Encapsulamento

## Exemplo Fazendeiro

Dentro da classe Lobo, há acesso ao atributo numeroDeOvelhas...

```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    Lobo lobo = new Lobo();

    lobo.ComerOvelha(pedro);
}
```

```
public bool ComerOvelha(Fazendeiro vitima)
{
    if (vitima.numeroDeOvelhas >= 1)
    {
        vitima.numeroDeOvelhas--;
        return true;
    }
    else
        return false;
}
```

Classe Lobo



lobo.ComerOvelha(pedro)



# Encapsulamento

## Revisão



- O encapsulamento permite que você controle o que sua classe compartilha com outras e o que ela mantém privado, de modo que nenhuma outra possa acessar.
- Existe uma forma fácil de evitar que atributos ou métodos internos de uma classe sejam acessados fora dela. Basta trocar a visibilidade desses membros de **public** para **private**.

```
class Fazendeiro
{
    private int numeroDeOvelhas = 10;
    public int sacosDeRacao;
    public const int multiplicadorDeRacao = 30;
}
```

# Encapsulamento

## Exemplo Fazendeiro

ERRO de compilação!  
O atributo numeroDeOvelhas é  
acessível apenas dentro da classe  
Fazendeiro.

```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    Lobo lobo = new Lobo();

    lobo.ComerOvelha(pedro);
}
```

Classe Lobo

```
public bool ComerOvelha(Fazendeiro vitima)
{
    if (vitima.numeroDeOvelhas >= 1)
    {
        vitima.numeroDeOvelhas--;
        return true;
    }
    else
        return false;
}
```



lobo.ComerOvelha(pedro)

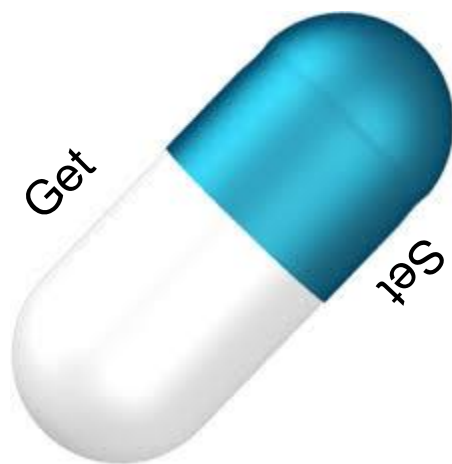


# Encapsulamento

## *Setters e Getters*



- Para permitir que atributos (ou métodos) privados sejam chamados de fora da classe, você pode criar métodos públicos que os utilizam. Dessa forma, o seu método estará controlando de forma **ADEQUADA** como esses membros internos devem ser usados!



- Por padrão, atributos privados costumam estar associados a métodos de modificação e acesso (*setters* e *getters*, respectivamente).

# Encapsulamento

## Exemplo Fazendeiro



```
class Fazendeiro
{
    private int numeroDeOvelhas = 10;
    public int sacosDeRacao;
    public const int multiplicadorDeRacao = 30;

    public int GetNumeroDeOvelhas()
    {
        return numeroDeVacas;
    }

    public bool SetNumeroDeOvelhas(int novoNumero)
    {
        if (novoNumero >= 0)
        {
            numeroDeOvelhas = novoNumero;
            sacosDeRacao = numeroDeOvelhas * multiplicadorDeRacao;
            return true;
        }
        else
            return false;
    }
}
```

# Encapsulamento

## Propriedades



- O C# possui tipos especiais de métodos que facilitam o processo de encapsular atributos.
- Você pode usar **propriedades**, métodos que realizam a tarefa dos métodos Setters e Getters resumindo a quantidade de código.



A melhor maneira de se entender seu funcionamento é observando a sua sintaxe.

# Código

## Propriedades



```
class Fazendeiro
{
    private int numeroDeOvelhas;
    public int SacosDeRacao;
    public const int MultiplicadorDeRacao = 30;

    public int NumeroDeOvelhas
    {
        get
        {
            return numeroDeOvelhas;
        }
        set
        {
            numeroDeOvelhas = value;
        }
    }
}
```

Declaração da propriedade



# Código

## Propriedades



```
class Fazendeiro
{
    private int numeroDeOvelhas;
    public int SacosDeRacao;
    public const int MultiplicadorDeRacao = 30;

    public int NumeroDeOvelhas
    {
        get
        {
            return numeroDeOvelhas;
        }
        set
        {
            numeroDeOvelhas = value;
        }
    }
}
```

Declaração da propriedade

Implementação do  
método Get

Implementação do  
método Set

# Código

## Propriedades



```
class Fazendeiro
{
    private int numeroDeOvelhas;
    public int SacosDeRacao;
    public const int MultiplicadorDeRacao = 30;
```

Declaração da propriedade

```
    public int NumeroDeOvelhas
    {
        get
        {
            return numeroDeOvelhas;
        }
        set
        {
            numeroDeOvelhas = value;
        }
    }
```

**value** é uma palavra reservada que funcionará como um parâmetro do método Set

# Usando a propriedade



```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    pedro.NumeroDeOvelhas = 10;

    int quantidadeDeOvelhas = pedro.NumeroDeOvelhas;
}
```

# Usando a propriedade



```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    pedro.NumeroDeOvelhas = 10;

    int quantidadeDeOvelhas = pedro.NumeroDeOvelhas;
}
```

Esse comando chama o método *setter* da propriedade.

# Usando a propriedade



```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    pedro.NumeroDeOvelhas = 10;

    int quantidadeDeOvelhas = pedro.NumeroDeOvelhas;
}
```

Esse comando chama o método *getter* da propriedade.

# Usando a propriedade



Colégio  
Pedro II

```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    pedro.NumeroDeOvelhas = 10;

    int quantidadeDeOvelhas = pedro.NumeroDeOvelhas;
}
```

Mas o comando não  
é o mesmo?



# Usando a propriedade



```
public void Main()
{
    Fazendeiro pedro = new Fazendeiro();

    2 pedro.NumeroDeOvelhas = 10;

    3 int quantidadeDeOvelhas = pedro.NumeroDeOvelhas;
}
```

Sim! Mas o contexto de uso é diferente! Repare que na linha (2) queremos modificar (setter) o valor do atributo `numeroDeOvelhas` enquanto que na linha (3) queremos recuperar o valor do atributo `numeroDeOvelhas`.

# Tratamento de Eventos

- GUIs são orientadas a eventos. Quando o usuário interage com algum controle da GUI, a interação – conhecida como evento – dirige o programa a realizar uma tarefa.
- Eventos comuns que fazem com que a aplicação reaja e execute uma tarefa incluem:
  - clicar em um **Button**;
  - escrever em uma **TextBox**;
  - selecionar um item de um **MenuBar**;
  - fechar uma janela;
  - mover o mouse;
- Todos os controles da GUI têm eventos associados com eles.
- Um método que executa alguma tarefa em resposta a um evento é chamado de *event handler* e o processo geral de responder a eventos é conhecido como *event handling*.



# Tratamento de Eventos

## Exemplo

The screenshot displays the Visual Studio IDE with a Windows Form named 'Form1' in Design view. The form contains a button labeled 'Chorar' and two text labels: 'Acabaram as férias' and 'Hahahaha'. The Properties window on the right shows the 'button1' control, which is of type 'System.Windows.Forms.Button'. The 'Ação' (Action) section is expanded, showing the 'Click' event with the handler 'button1\_Click'.

Form1.cs   Program.cs   Form1.cs [Design]   ▢   ✕

Form1

Acabaram as férias

Hahahaha

Chorar

Propriedades   ▾   🔍   ✕

**button1** System.Windows.Forms.Button

▢ **Ação**

Click	button1_Click
MouseCaptureChanged	
MouseClicked	

▢ **Aparência**

Paint	
-------	--

▢ **Arrastar e Soltar**

DragDrop	
DragEnter	
DragLeave	
DragOver	
GiveFeedback	
QueryContinueDrag	

▢ **Chave**

KeyDown	
KeyPress	
KeyUp	
PreviewKeyDown	

▢ **Comportamento**

ChangeUICues	
--------------	--

# Tratamento de Eventos

## Exemplo

The screenshot shows the Visual Studio IDE with the 'Form1.cs [Design]' window active. The form contains a button labeled 'Chorar'. A blue arrow points from the button to the 'button1\_Click' event in the 'Ação' section of the 'Propriedades' window. The 'Propriedades' window shows the following sections:

- Ação**
  - Click: **button1\_Click**
  - MouseCaptureChanged
  - MouseDown
- Aparência**
  - Paint
- Arrastar e Soltar**
  - DragDrop
  - DragEnter
- Chave**
  - KeyDown
  - KeyPress
  - KeyUp
  - PreviewKeyDown
- Comportamento**
  - ChangeUICues

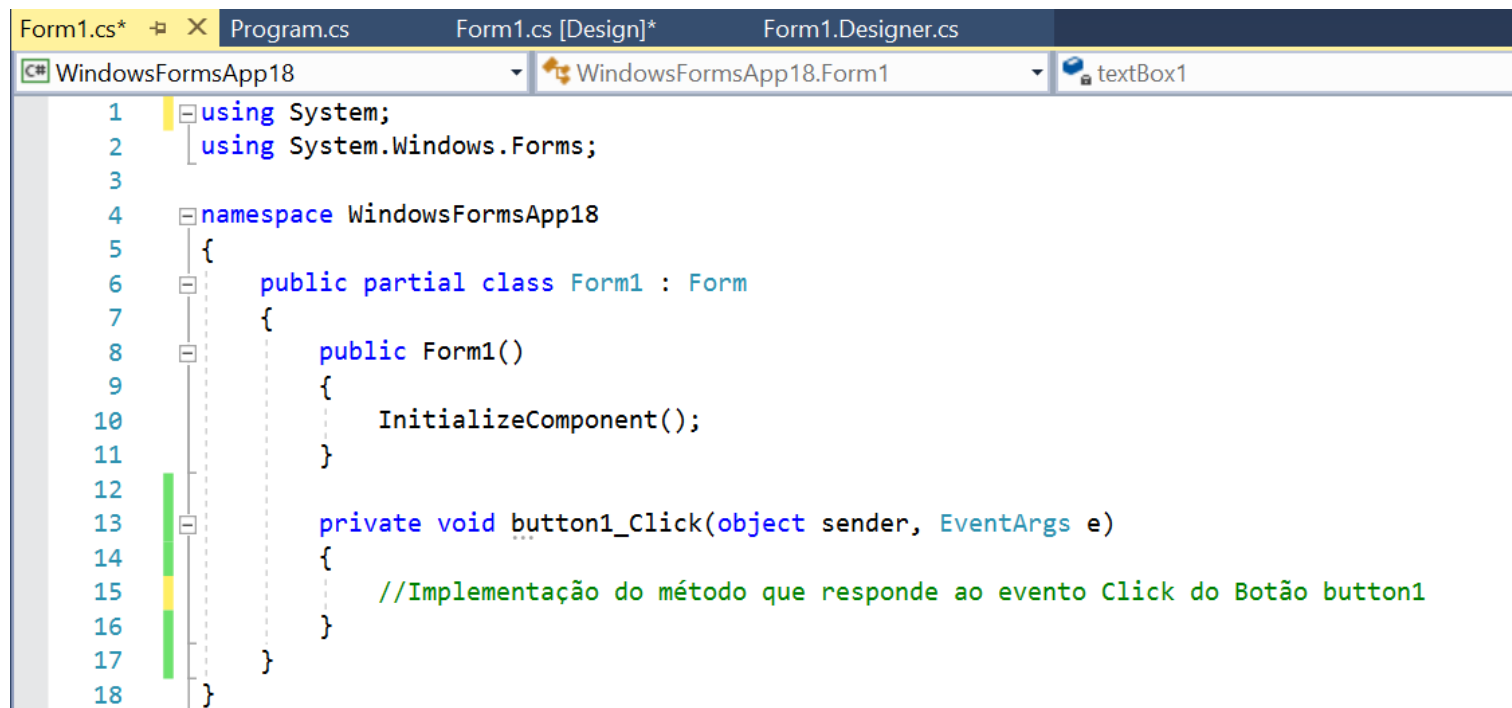
O método denominado `button1_Click` irá tratar do evento Click que o controle `button1` pode gerar.

# Tratamento de Eventos

## Exemplo

- Por convenção, o VS nomeia o método **tratador de evento** como:

nomeDoObjeto\_NomeDoEvento (object sender, EventArgs e).



```
1  using System;
2  using System.Windows.Forms;
3
4  namespace WindowsFormsApp18
5  {
6      public partial class Form1 : Form
7      {
8          public Form1()
9          {
10             InitializeComponent();
11         }
12
13         private void button1_Click(object sender, EventArgs e)
14         {
15             //Implementação do método que responde ao evento Click do Botão button1
16         }
17     }
18 }
```

- O **event handler** button1\_Click executa quando o usuário clica no controle clickButton.

# Tratamento de Eventos

## Outras possibilidades

- Vários controles podem disparar vários eventos. Ao clicarmos na aba de Eventos, podemos escolher que tipo de tratador gostaríamos de criar como método em nosso formulário. A tabela abaixo mostra alguns controles, eventos e tratadores.

Controle	Evento	Método Tratador do Evento
Button	Click	button_Click(object sender, EventArgs e)
TextBox	TextChanged	textBox_TextChanged(object sender, EventArgs e)
Label	Click	label_Click(object sender, EventArgs e)
NumericUpDown	ValueChanged	numUpDown_ValueChanged(object sender, EventArgs e)

# Mais controles

- Iremos dar uma olhada em diversos outros controles que podem ser adicionados nos formulários.
- Para isso, iremos estudar quais são as suas propriedades, seus métodos e seus possíveis eventos que podem ser capturados e implementados através de um tratador de eventos.

# Labels, TextBoxes e Buttons

- **Labels** proveem informação textual (ou também opcionalmente imagens) e são definidos através da classe Label. Um label apresenta um texto que o usuário não pode modificar diretamente.

Propriedades da Classe Label	Descrição
Text	Especifica o texto associado ao Label

Eventos da Classe Label	Descrição
Click	É gerado quando a CheckBox é clicada. Esse é o evento padrão das Labels. Quando o usuário clica duas vezes na Label no Designer, um tratador de evento para este evento é criado no código.

# Labels, TextBoxes e Buttons

- **TextBoxes** (classe TextBox) é uma área onde texto pode ser exibido por um programa ou o usuário pode usá-la para passar informações através do teclado.

Propriedades da Classe TextBox	Descrição
Text	Especifica o texto associado ao Label
ReadOnly	Se true, a TextBox terá um fundo cinza e seu texto não poderá ser editado.
UseSystem-PasswordChar	Se true, a TextBox irá esconder as informações que estejam na propriedade Text apresentando asteriscos.

Eventos da Classe TextBox	Descrição
TextChanged	É gerado quando o texto muda na TextBox (quando o usuário adiciona ou remove caractere). Esse é o evento padrão das TextBoxes. Quando o usuário clica duas vezes na TextBox no Designer, um tratador de evento para este evento é criado no código.

# Labels, TextBoxes e Buttons

- Um botão é um controle que o usuário clica para disparar uma ação no programa. São implementados através da classe **Button**.

Propriedades da Classe Button	Descrição
Text	Especifica o texto exibido na face do botão.
Enabled	Se true, o botão será desabilitado.

Eventos da Classe Button	Descrição
Click	É gerado quando o botão é clicado. Esse é o evento padrão das Button. Quando o usuário clica duas vezes na Button no Designer, um tratador de evento para este evento é criado no código.



# NumericUpDown

- Pode ser necessário criar uma caixa de entrada que permita apenas que o usuário digite valores numéricos.
- Não somente, pode ser necessário também especificar apenas um intervalo de valores numéricos.
- Um controle **NumericUpDown** apresenta a mesma ideia do comando TextBox, mas permite apenas que o usuário entre com números.
- Não somente, pode ser necessário também especificar apenas um intervalo de valores numéricos.



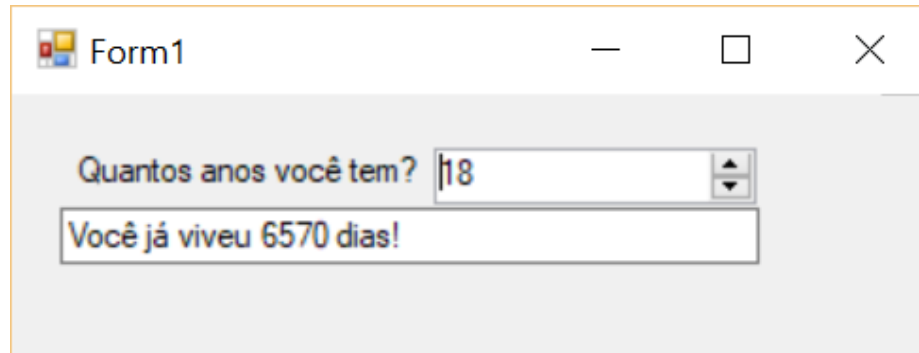
# NumericUpDown

- A lista seguir mostra propriedades, métodos e eventos comuns a classe NumericUpDown.

Propriedades da Classe NumericUpDown	Descrição
DecimalPlaces	Especifica quantas casas decimais serão aceitas.
Increment	Especifica qual será o incremento da NumericUpDown sempre que a setinha for clicada.
Maximum	Determina qual o maior valor que a NumericUpDown pode assumir.
Minimum	Determina qual o menor valor que a NumericUpDown pode assumir.
Value	O valor numérico que está sendo exibido. Corresponde ao Text da TextBox.

Eventos da Classe NumericUpDown	Descrição
ValueChanged	Esse evento é disparado quando o valor da NumericUpDown muda. Esse é o evento padrão da NumericUpDown. Quando o usuário clica duas vezes na NumericUpDown no Designer, um tratador de evento para este evento é criado no código.

# NumericUpDown

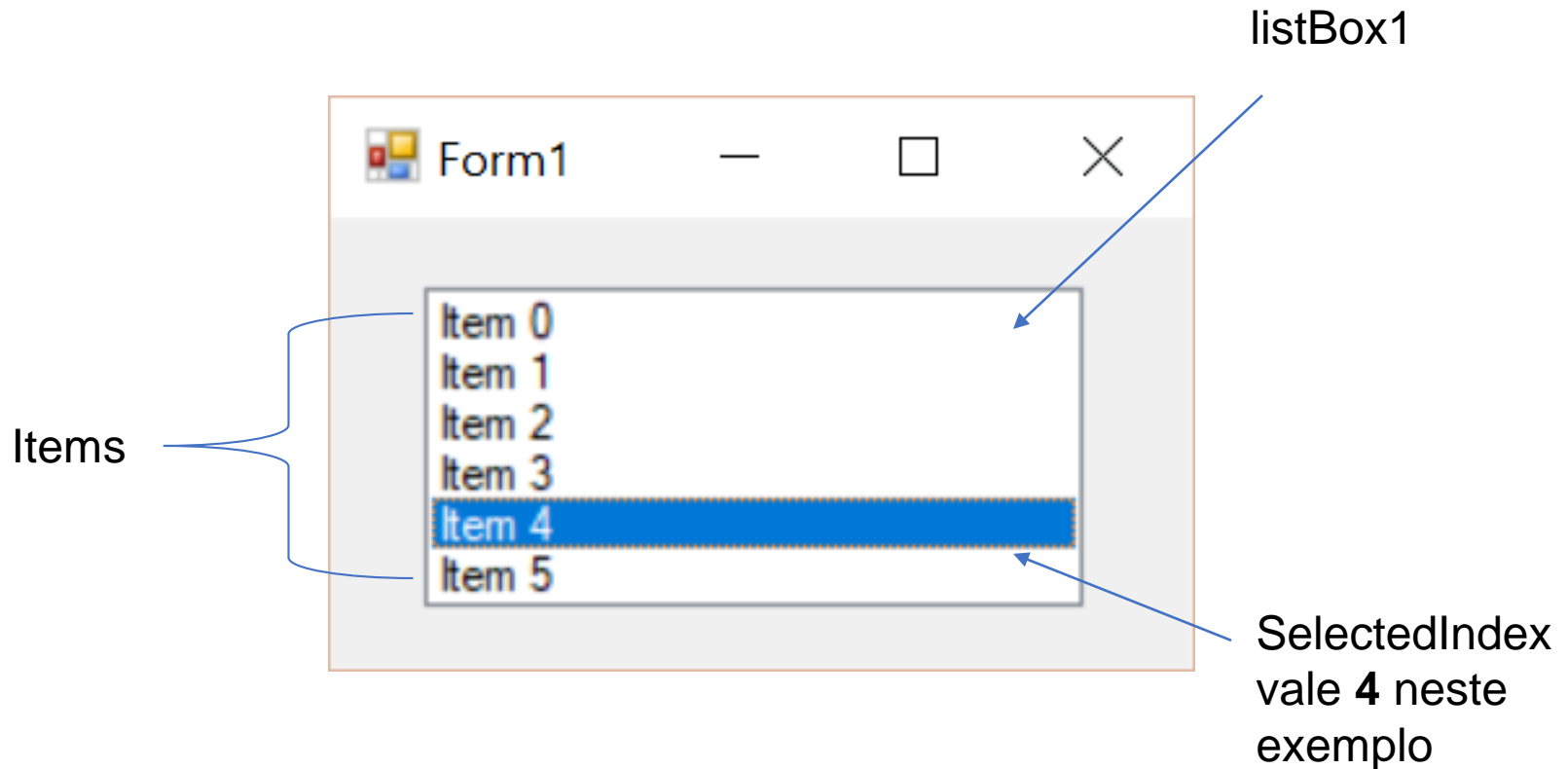


```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
    decimal dias = numericUpDown1.Value * 365;
    string strDias = Convert.ToString(dias);

    textBox1.Text = "Você já viveu " + strDias + " dias!";
}
```

# List Box

- Suponha que o objeto da ListBox neste formulário é denominado listBox1.



# List Box

## Trabalhando com a propriedade Items

- Através da propriedade Items nós temos acesso a cada elemento na lista.
- A propriedade Items nada mais é do que um objeto do tipo List. Sendo assim, todos os métodos e propriedades que aprendemos a utilizar da classe List podem ser utilizados!

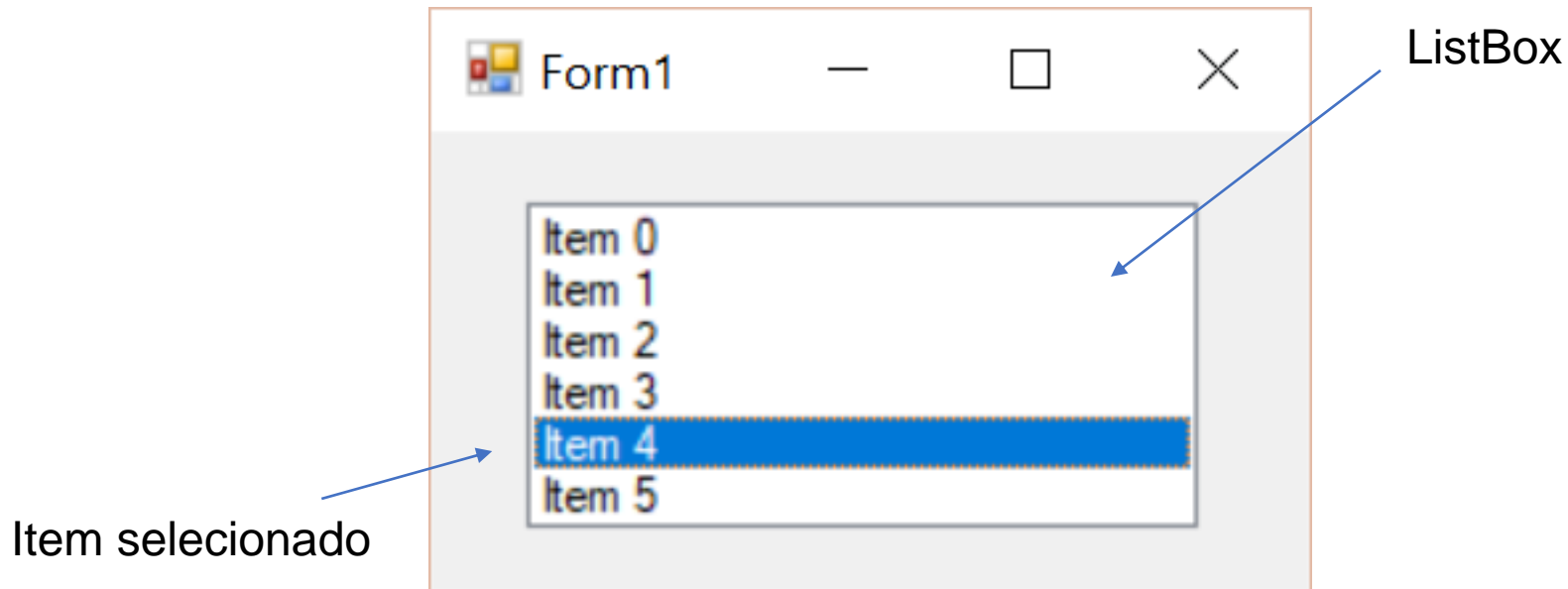


### Exemplo

```
Count  
Add()  
RemoveAt()  
Clear()
```

# List Box

- O controle **ListBox** permite que o usuário veja e selecione um item de uma lista repleta de itens.



# List Box

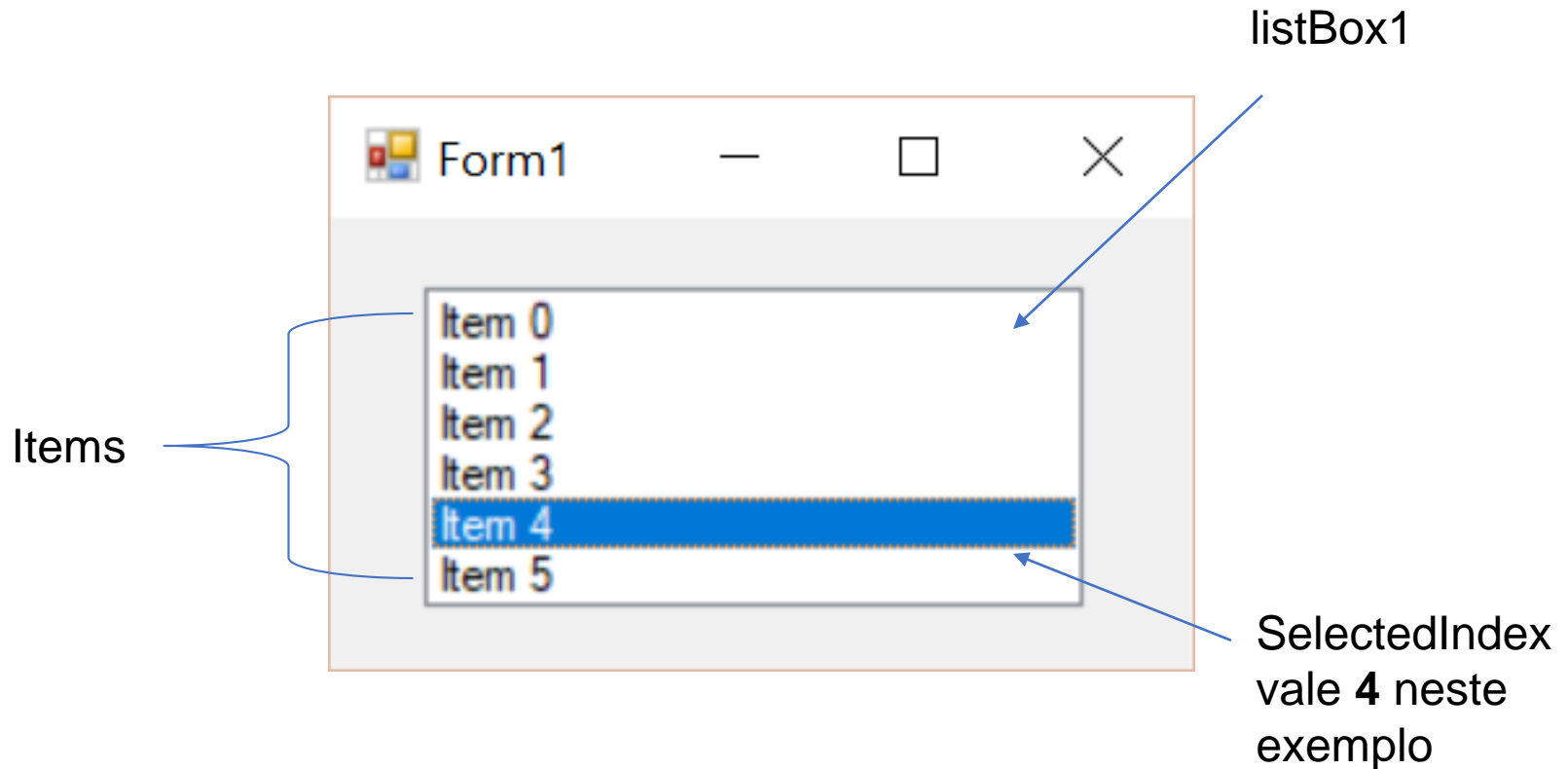
- A lista seguir mostra propriedades, métodos e eventos comuns a classe ListBox.

Propriedades da Classe ListBox	Descrição
Items	Representa a coleção de itens na ListBox.
SelectedIndex	Retorna o índice do item selecionado. Se nenhum item está selecionado, a propriedade retorna -1.
SelectedItem	Retorna uma referência para o item selecionado.

Eventos da Classe ListBox	Descrição
SelectedIndexChanged	É gerado quando o item selecionado na lista muda. Esse é o evento padrão das ListBoxes. Quando o usuário clica duas vezes na ListBox no Designer, um tratador de evento para este evento é criado no código.

# List Box

- Suponha que o objeto da ListBox neste formulário é denominado listBox1.





# List Box

## Trabalhando com a propriedade Items

- Através da propriedade Items nós temos acesso a cada elemento na lista.
- A propriedade Items nada mais é do que um objeto do tipo List. Sendo assim, todos os métodos e propriedades que aprendemos a utilizar da classe List podem ser utilizados!



### Exemplo

```
Count  
Add()  
RemoveAt()  
Clear()
```

# List Box

## Adicionando Itens

- Para adicionar novos itens na ListBox nós devemos adicionar objetos para sua coleção de itens.
- Primeiro acessamos a propriedade Items da ListBox (que é um objeto do tipo List) e depois chamamos o método Add.

`listBox1.Items.Add("Tafare1");`

ListBox

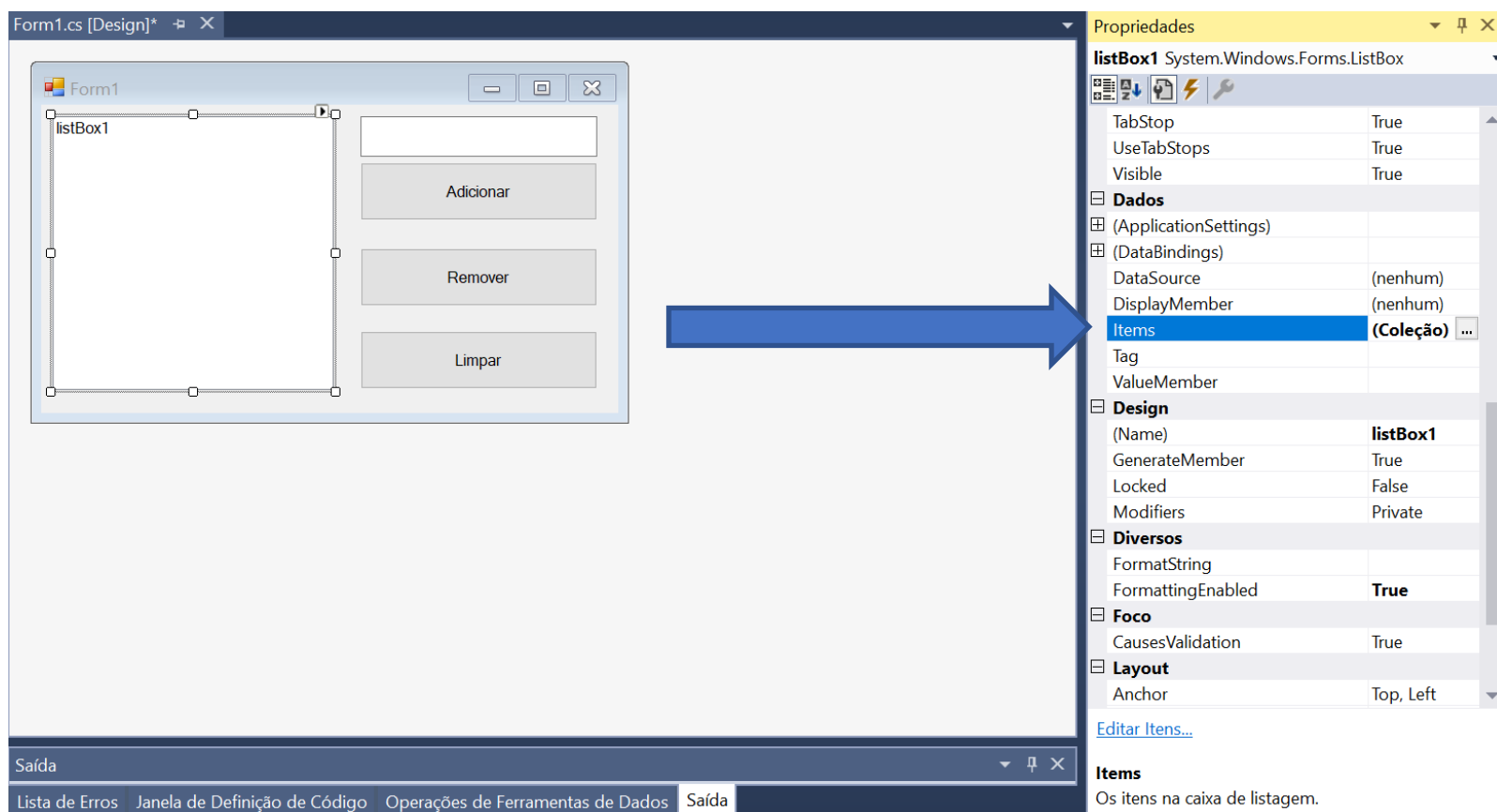
List

Método da classe List

# List Box

## Adicionando Itens

- Alternativamente, você pode adicionar novos itens à lista através da janela de Designer acessando a propriedade Items de forma direta.



# List Box

## Adicionando Itens

- Para adicionar novos itens na ListBox nós devemos adicionar objetos para sua coleção de itens.
- Primeiro acessamos a propriedade Items da ListBox (que é um objeto do tipo List) e depois chamamos o método Add.

`listBox1.Items.Add("Tafare1");`

ListBox

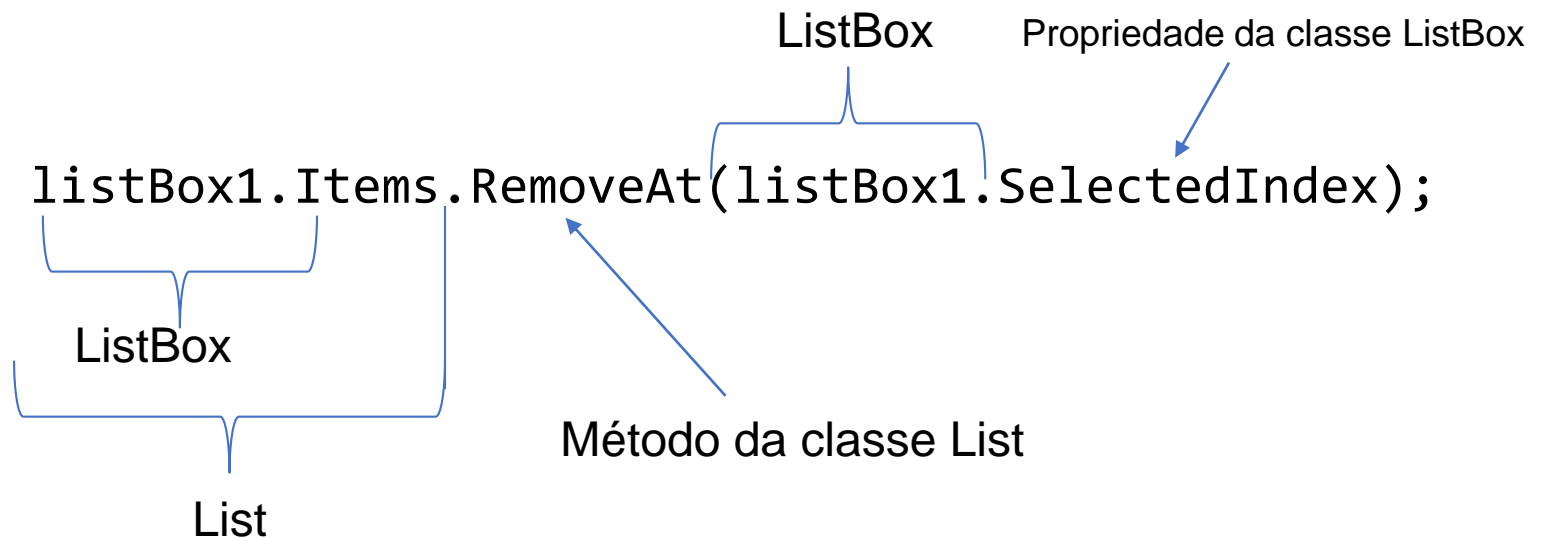
List

Método da classe List

# List Box

## Removendo Itens

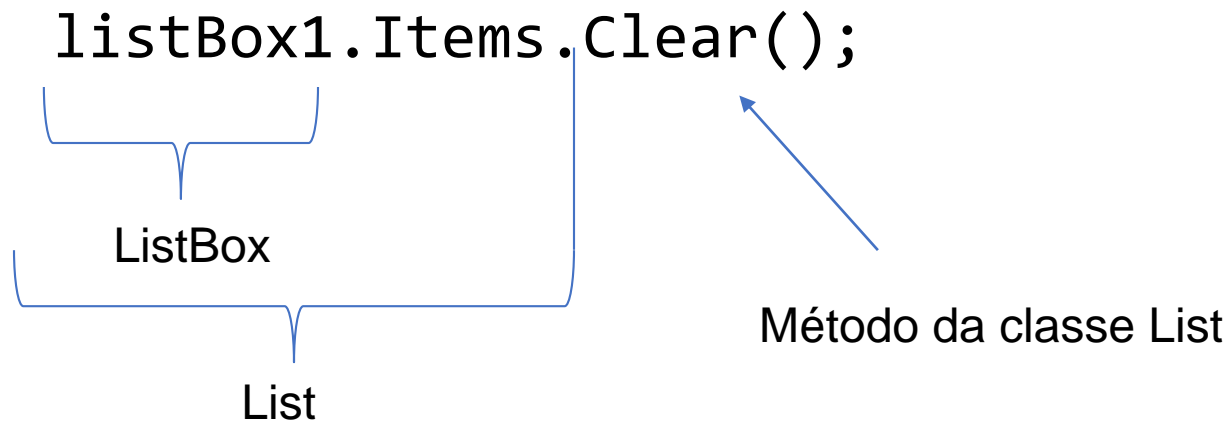
- Para remover um item da ListBox, use o método RemoveAt e passe como parâmetro para ele o índice do item que está sendo selecionado.



# List Box

## Limpendo os Itens

- Para limpar todos os itens da lista, basta chamar o método `Clear` da classe `List`.



# List Box

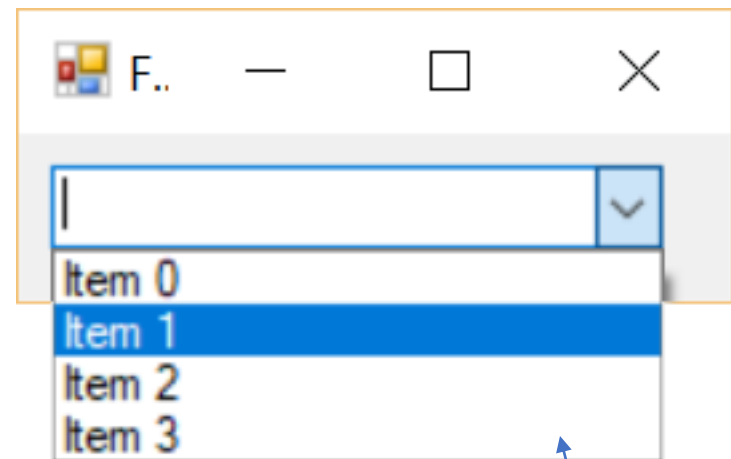
## Exercício

- Implemente o Formulário abaixo que realize todas as operações sinalizadas nos botões.

The image shows a Windows application window titled "Form1". Inside the window, on the left, is a list box containing three items: "João", "Flávio", and "Cláudio". The item "Flávio" is currently selected, highlighted in blue. To the right of the list box, there is a vertical stack of controls. At the top is an empty text input field. Below it are three buttons: "Adicionar", "Remover", and "Limpar", arranged vertically. The buttons are light gray with black text.

# ComboBox

- O controle **ComboBox** combina funcionalidades de TextBox com funcionalidades de drop-down list.
- Uma ComboBox geralmente aparece como uma TextBox com uma setinha para baixo a sua direita. Por padrão, o usuário pode inserir informações na ComboBox ou clicar na setinha para ver uma lista pré-definida de itens.
- Assim como a ListBox, na ComboBox você pode acessar uma propriedade Items e adicionar itens através do método Add, remover itens através do método RemoveAt, etc.



ComboBox



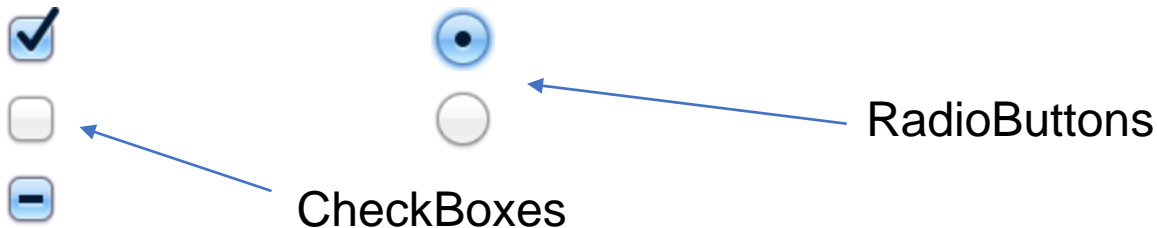
# ComboBox

- Algumas propriedades, métodos e eventos da classe ComboBox são listados.

Propriedades da Classe ComboBox	Descrição
Items	Representa a coleção de itens na ComboBox.
SelectedIndex	Retorna o índice do item selecionado. Se nenhum item está selecionado, a propriedade retorna -1.
Sorted	Indica se os itens estão ordenados alfabeticamente. Ao atribuir true a esta propriedade ordena os itens. O padrão é false.
Eventos da Classe ComboBox	Descrição
SelectedIndexChanged	É gerado quando o item selecionado na lista muda. Esse é o evento padrão das ComboBoxes. Quando o usuário clica duas vezes na ComboBox no Designer, um tratador de evento para este evento é criado no código.

# RadioButton e CheckBox

- C# tem dois tipos de botões de estado que podem estar no estado de on/off ou true/false. São eles: **CheckBoxes** e **RadioButtons**, respectivamente.



# RadioButtons

- Botões de rádio (definidos pela classe `RadioButton`) são semelhantes às `CheckBoxes`: ambas as classes compartilham da ideia de ter dois estados.

No entanto, `RadioButtons` normalmente aparecem em grupos, onde **apenas um Botão pode estar selecionado dentro do grupo**.

- Para dividir `RadioButtons` em vários grupos, eles devem ser adicionados em containers separados, como por exemplo a classe `GroupBox`.

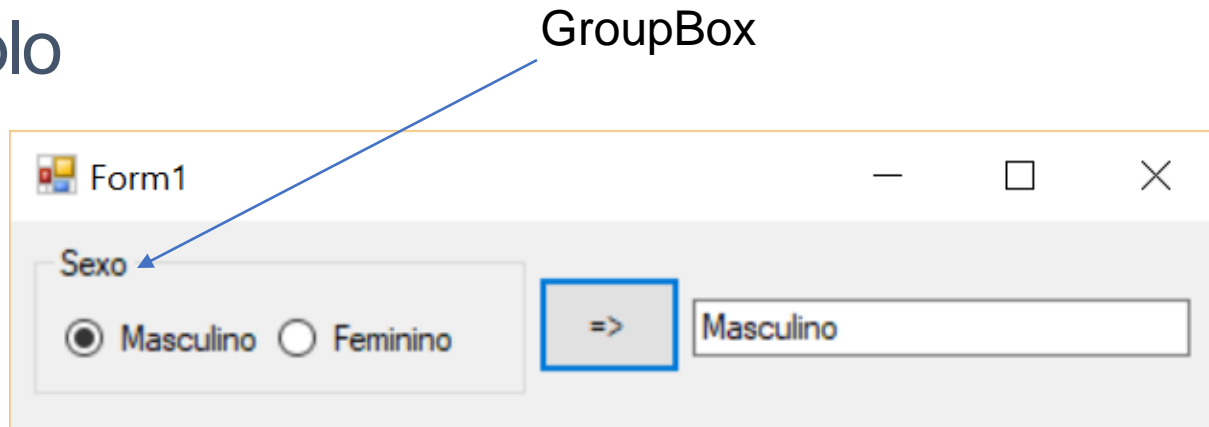
# RadioButtons

Propriedades da Classe CheckBox	Descrição
Checked	Indica se o RadioButton está marcado, retornando true ou não, retornando false.
Text	Especifica o texto associado ao RadioButton

Eventos da Classe CheckBox	Descrição
CheckedChanged	É gerado quando a CheckBox muda sua marcação. Isso é o evento padrão das CheckBoxes. Quando o usuário clica duas vezes a CheckBox no Designer, um tratador de evento para este evento é criado no código.

# RadioButtons

## Exemplo



```
private void button1_Click(object sender, EventArgs e)
{
    if (radioButton1.Checked)
        textBox1.Text = radioButton1.Text;

    if (radioButton2.Checked)
        textBox1.Text = radioButton2.Text;
}
```

# CheckBox

- CheckBox é um quadrado pequeno que ou está vazio ou contém uma marca de “check”.
- Quando o usuário clica numa CheckBox, uma marca de “check” aparece. Ao se clicar uma segunda vez, a marca desaparece.

**Qualquer número de CheckBoxes pode estar selecionado ao mesmo tempo.**

# CheckBox

Propriedades da Classe CheckBox	Descrição
Checked	Indica se a CheckBox está marcada retornando true ou false, caso contrário.
Text	Especifica o texto associado a CheckBox

Eventos da Classe CheckBox	Descrição
CheckedChanged	É gerado quando a CheckBox muda sua marcação. Isso é o evento padrão das CheckBoxes. Quando o usuário clica duas vezes a CheckBox no Designer, um tratador de evento para este evento é criado no código.