

# Capítulo 6

## C Sharp

---

- Programação
- Tipos de dados e variáveis
- Operadores
- Estrutura de decisão
- Estruturas de repetição usadas na linguagem
- Tratamento de erros / exceções
- Vetores e matrizes
- Classes
- Windows Form Application - componentes
- Eventos

O engenheiro de software Anders Hejlsberg nasceu na Dinamarca, na cidade de Copenhagem, em 1960. Um importante marco em sua carreira foi ter escrito os compiladores Pascal e Blue Label, quando trabalhava na Nascon Computer. Foi, porém, mais tarde, para a Borland que desenvolveu o Turbo Pascal e o Delphi, antes de ser contratado pela Microsoft, em 1996. Criou a linguagem J++ e a C#, em cujo desenvolvimento trabalha continuamente.

Considerada como a mais importante linguagem de desenvolvimento da Microsoft dentro da Plataforma .NET Framework, a C# vem ganhando inúmeros adeptos (programadores de outras linguagens) em virtude de sua característica e semelhança ao C, C++ ou Java. A linguagem de programação C# é orientada a objetos e foi criada praticamente a partir do zero para compor a plataforma. Trata-se do primeiro compilador com suporte de programação para a maioria das classes do .NET Frameworks. Embora tenha sido feita por vários programadores, os méritos são atribuídos principalmente a **Anders Hejlsberg**, muito conhecido por desenvolvedores do compilador Delphi.

## 6.1. Programação

O C# requer toda a lógica de programação contida em classes. Neste capítulo, não serão abordados, em detalhes, os conceitos de programação orientada a objetos. Caso haja dúvidas consulte o capítulo 4.

### 6.1.1. Console Application

A aplicação do tipo Console Application (aplicação do painel de controle em português) se assemelha às programações da linguagem C/C++. Seu resultado é apresentado na janela de Prompt do DOS, na qual programamos diretamente pelo método Main(), só que, agora, no formato de uma classe (figura 217).

Pode-se fazer uma analogia ao Java: no começo do nosso código, encontramos os using's (import), a classe principal Program e o método Main() com a implementação do código em C#.

### 6.1.2. Windows Form Application

No Windows Form Application (aplicação de formulários de Windows), a estrutura do projeto é diferente do Console Application. Se analisarmos a janela do Solution Explorer, podemos verificar que a disposição dos códigos têm arquivos separados, com funções específicas, como se pode ver em seguida.

**Program.cs:** possui o método Main() (figura 218), que executa o primeiro formulário (Form1). Nesse caso, EnableVisualStyles() define o estilo visual do

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Meu primeiro programa em C#");
            Console.ReadKey();
        }
    }
}
```

**Figura 217**  
Console Application.

projeto, SetCompatibleTextRenderingDefault(), o modo e propriedade como os componentes serão visualizados, e o Run(), para execução do formulário.

**Form1.Designer.cs:** realiza a especificação dos componentes utilizados na aplicação (figura 219). Quando um projeto começa a ser construído, toda a estrutura dos seus componentes (tamanho, fonte, nome, localização e outras propriedades) fica registrada no formulário, seguindo o método chamado InitializeComponent(), exatamente no qual encontramos a expressão “Windows Form Designer generated code”.

**Form1.cs:** reapresenta a maior parte da nossa programação em relação aos componentes e aos eventos (figura 220). Insere-se, então, a lógica de programação, ou seja, aquilo que deve ser feito ao inicializar um formulário (Form1\_Load), quando o usuário clicar em um botão e em outros procedimentos do projeto.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Exemplo01
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

**Figura 218**  
Program.cs.

**Figura 219**  
Form1.Designer.cs.

```
namespace Exemplo01
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Windows Form Designer generated code
    }
}
```

**Figura 220**  
Form1.cs.

```
namespace Exemplo01
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            MessageBox.Show("Meu Primeiro programa para Windows");
        }
    }
}
```

## 6.2. Tipos de dados e variáveis

Os tipos de dados em C# são distribuídos da forma como apresenta a tabela 13.

Utilizamos as variáveis para armazenar diferentes tipos de dados (números, palavras, data etc.), como já foi mostrado antes. Para a criação de uma variável, deve-se dar um nome único, que identifique o seu conteúdo (consulte o quadro *Recomendações para a boa prática*).

TIPO	IMPLEMENTAÇÃO
Byte	Inteiro de 8 bits sem sinal (0 a 255).
Sbyte	Inteiro de 8 bits com sinal (-127 a 128).
Ushort	Inteiro de 16 bits sem sinal (0 a 65 535).
Short	Inteiro de 16 bits com sinal (-32 768 a 32 767).
UInt	Inteiro de 32 bits sem sinal (0 a 4 294 967 295).
Int	Inteiro de 32 bits com sinal (-2 147 483 648 a 2 147 483 647).
Ulong	Inteiro de 64 bits sem sinal (0 a 18 446 744 073 709 551 615).
Long	Inteiro de 64 bits com sinal (-9 223 372 036 854 775 808 a 9 223 372 036 854 775 807).
Double	Ponto flutuante binário IEEE de 8 bytes (±5.0 _ 10-324 a ±1.7 _ 10308), 15 dígitos decimais de precisão.
Float	Ponto flutuante binário IEEE de 4 bytes (±1.5 _ 10-45 a ±3.4 _ 1038), 7 dígitos decimais de precisão.
Decimal	Ponto flutuante decimal de 128 bits. (1.0 _ 10-28 a 7.9 _ 1028), 28 dígitos decimais de precisão.
Bool	Pode ter os valores true e false. Não é compatível com inteiro.
Char	Um único caractere Unicode de 16 bits. Não é compatível com inteiro.

**Tabela 13**  
Distribuição dos tipos de dados.

O importante é fazer corretamente as declarações das variáveis e os tipos de dados aos quais elas pertencem, como está exemplificado na figura 221.

```
int num;
int x, y, x;
float salario;
string nome;
string edereco, cidade, estado;
char resp;
```

**Figura 221**  
Declarações corretas das variáveis.

### RECOMENDAÇÕES PARA A BOA PRÁTICA

- Se houver mais de uma palavra, a primeira letra da segunda deve ser maiúscula:

```
nomeCliente
pesoMedio
idadeMaxima
```

- Pode-se usar um prefixo para expressar, no início da variável, o tipo de dado ao qual ela pertence:

```
strNomeCliente
floPesoMedio
intIdadeMaxima
```

#### 6.2.1. Alocação de memória

A alocação e a utilização de variáveis de memória podem ser realizadas de duas maneiras: Stack (pilha) e Heap (amontoado). A diferença entre elas está na forma como cada uma trata as informações.

**Stack:** Nessa área, ficam as variáveis locais ou os parâmetros de funções de valor ou referência. Sua limpeza é realizada automaticamente na saída de cada função, usando return ou não.

### ATENÇÃO:

No momento da programação, é fundamental ficar alerta para ver onde o código será inserido. Portanto, identifique corretamente o componente e o evento que será agregado. As instruções básicas de programação serão apresentadas, aqui, utilizando-se o método Console Application, pois a sua estrutura é a mesma para aplicações gráficas do Windows Form Application.

**Heap:** Aqui, ficam os valores referentes aos objetos. Isso ocorre quando uma classe é estabelecida usando o new (construtores). Caso as funções sejam encerradas ou ocorra um exception, a limpeza não acontece automaticamente e sim por meio do Garbage Collector.

### 6.3. Operadores

Exemplos práticos para mostrar a funcionalidade dos operadores são executados no Console Application, como já foi mencionado.

#### 6.3.1. Operadores aritméticos

Para visualizar o efeito dos operadores (indicados no quadro *Aritméticos*), utilize a classe Console(), a qual, pelo método WriteLine(), revela uma expressão ou um conteúdo de uma variável com a quebra de linha automática. Já o Write() realiza a impressão sem a quebra de linha.

ARITMÉTICOS	
+	Adição
-	Subtração
*	Multipliação
/	Divisão
%	Resto da divisão

#### 6.3.2. Operadores relacionais

RELACIONAIS	
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
==	Igual
!=	Diferente

#### 6.3.3. Operadores aritméticos de atribuição reduzida

ARITMÉTICOS	
+=	Adição Igual
-=	Subtração Igual
*=	Multipliação Igual
/=	Divisão Igual
%=	Resto da divisão Igual

Outro método da classe Console() utilizado neste exemplo é o Readkey(), que permite ao programa realizar uma pausa, aguardando até que uma tecla seja pressionada. Isso é necessário por estarmos trabalhando com uma janela DOS, que se encerra quando o programa é finalizado (figura 222).

Observe sua aplicação na figura 223.

```
static void Main(string[] args)
{
    Console.WriteLine("Verificando os Operadores");
    int x = 10;
    int y = 15;
    Console.WriteLine("Soma: " + (x + y));
    Console.WriteLine("Subtração: " + (x - y));
    Console.WriteLine("Multipliação: " + (x * y));
    Console.WriteLine("Divisão: " + (y / x));
    Console.WriteLine("Divisão (10/3): " + (x / 3));
    Console.WriteLine("Resto da Divisão (10/3): " + (x % 3));
    Console.ReadKey();
}
```

**Figura 222**  
Usando o método Readkey.

```
static void Main(string[] args)
{
    Console.WriteLine("Operadores Reduzidos");
    int x = 10;
    int y = 15;
    Console.WriteLine("Soma + igual: " + (x += 2));
    Console.WriteLine("Subtração + igual: " + (y -= 10));
    // x está com novo valor !!!
    Console.WriteLine("Multipliação + igual: " + (x *= 2));
    Console.WriteLine("Divisão + igual: " + (x /= 2));
    Console.WriteLine("Resto da Divisão + igual: " + (x %=5));
    Console.ReadKey();
}
```

**Figura 223**  
Aplicação de operadores aritméticos de atribuição reduzida.

#### 6.3.4. Operadores de incremento e decremento

OPERADORES	
++	Incremento
--	Decremento

Observe sua aplicação na figura 224.

```
static void Main(string[] args)
{
    Console.WriteLine("Operadores Reduzidos");
    int x = 10;
    int y = 10;
    x++;
    Console.WriteLine("Incremento:" + x);
    y--;
    Console.WriteLine("Decremento:" + y);
    Console.ReadKey();
}
```

**Figura 224**  
Aplicação de operadores de incremento e decremento.

### 6.3.5. Operadores lógicos

OPERADORES	
&&	And
	Or
!	Not

Sua aplicação é ilustrada na figura 225.

**Figura 225**  
Aplicação de operadores lógicos.

```
static void Main(string[] args)
{
    Console.WriteLine("Operadores Lógicos");
    int a = 10;
    int b = 30;
    int c = 10;
    int d = 25;
    Console.WriteLine((a < d) && (b != c));
    Console.WriteLine((a > d) || (b != c));
    Console.WriteLine(! (a >= b));
    Console.ReadKey();
}
```

### 6.3.6. Conversões C#

Para realizar as conversões de dados, usamos os mesmos conceitos trabalhados em Java. No entanto, dependendo da instrução adotada, o resultado obtido poderá variar. Um bom programador realiza alguns testes para verificar qual a melhor maneira de fazer a conversão.

### 6.3.7. Parse

A classe Parse sempre vem precedida do tipo de dados e em seguida da variável ou da expressão a ser convertida (figura 226).

**Figura 226**  
Classe Parse.

```
int.Parse(idade);
float.Parse(salario);
int.Parse(Console.ReadLine());

valor = Int32.Parse(numero);
teste = Int16.Parse(dado);

dt = DateTime.Parse("01/01/2010");
```

### 6.3.8. Convert

A classe Convert tem a sua indicação de tipo registrada após o nome da classe, seguido da variável ou da expressão (figura 227).

```
Convert.ToUInt16(indade);
Convert.ToDouble(salario);
Convert.ToInt16(Console.ReadLine());

valor = Convert.ToInt32(numero);
teste = Convert.ToInt16(dado);

dt = Convert.ToDateTime("01/01/2010");
```

**Figura 227**  
Classe Convert.

**DICA**  
Existe um grande número de métodos da Classe Convert e Parse. Para conhecê-los, consulte o Msdn .NET Frameworks Developer Center. <http://msdn.microsoft.com/pt-br/library/default.aspx>. Os próximos tópicos envolvem os conhecimentos prévios de lógica de programação. Caso haja dúvidas, recorra aos conceitos tratados anteriormente.

## 6.4. Estrutura de decisão

Assim como o Java, os controles de início e fim de cada estrutura deverão ser controlados pelos "{}". Para realizar os desvios condicionais, utilizamos a estrutura if() ou switch(), traduzidos como "se" e "troca", respectivamente.

### 6.4.1. Condição verdadeiro – if

A mesma recomendação feita para Java vale para o C#. Ou seja, quando a instrução if() apresentar uma única instrução para a sua condição, não é necessário utilizar as chaves, que se tornam opcionais. No exemplo (figura 228), podemos verificar que a variável "x" é maior que o valor "10", sabendo-se que o seu valor inicial é "5". E visualizamos a expressão: "A variável X é maior que 10".

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura IF");
    int x = 15;
    if (x > 10) {
        Console.WriteLine("A variável X é maior que 10");
    }
    Console.ReadKey();
}
```

**Figura 228**  
Exemplo de instrução if().

### 6.4.2. Condição verdadeiro ou falso – if...else

Já nesse outro exemplo (figura 229), verificaremos se a variável "x" é maior que "10" ou não, sabendo-se que o seu valor inicial é "5". Será adicionada uma expressão para cada alternativa.

**Figura 229**

Exemplo de if...else.

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura IF");
    int x = 5;
    if (x > 10)
    {
        Console.WriteLine("A variável X é maior que 10");
    }
    else {
        Console.WriteLine("A variável X é menor que 10");
    }
    Console.ReadKey();
}
```

### 6.4.3. Condições múltiplas – if...elseif...elseif...else

Agora, verificaremos se a variável “x” possui o número 1, 2 ou 3, sabendo-se que o valor inicial é 3. Nesse caso, para qualquer outro valor, será visualizada a expressão: “O valor de X é TRÊS” (figura 230).

**Figura 230**

Exemplo de condições múltiplas.

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura IF");
    int x = 3;
    if (x ==1)
    {
        Console.WriteLine("O valor de X é UM");
    }
    else if (x==2) {
        Console.WriteLine("O Valor de X é DOIS");
    }
    else if(x==3){
        Console.WriteLine("O Valor de X é TRÊS");
    }
    else {
        Console.WriteLine("Qualquer outro valor");
    }
    Console.ReadKey();
}
```

### 6.4.4. Múltiplos testes – Switch()

Usando o mesmo exemplo do item anterior, a sequência de testes é realizada com a instrução switch(). Assim, cada um será implementado com a instrução break para que as outras condições não sejam executadas (figura 231). A instrução default está realizando a função da instrução else do if().

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura SWITCH");
    int x = 3;
    switch (x) {
        case 1:
            Console.WriteLine("O valor de X é UM");
            break;
        case 2:
            Console.WriteLine("O valor de X é DOIS");
            break;
        case 3:
            Console.WriteLine("O valor de X é TRÊS");
            break;
        default:
            Console.WriteLine("Qualquer outro valor");
            break;
    }
    Console.ReadKey();
}
```

**Figura 231**

Exemplo de múltiplos testes – Switch().

## 6.5. Estruturas de repetição usadas na linguagem

### 6.5.1. While()

Usando a variável “cont”, para controle do loop, obtemos os números de 0 a 10. O importante em uma instrução While() é a implementação de um contador dentro da estrutura (figura 232).

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura WHILE");
    int cont=0;
    while (cont <=10){
        Console.WriteLine("Numero: “ + cont);
        cont++;
    }
    Console.ReadKey();
}
```

**Figura 232**

Exemplo de While.

### 6.5.2. Do... While()

Nesse caso, repete-se a instrução para While(). Porém, o teste é realizado no final do loop. Esse tipo de estrutura permite que as instruções que estão dentro do laço de repetição sejam executadas, no mínimo, uma vez (figura 233).

**Figura 233**

Do... While().

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura DO...WHILE");
    int cont=0;
    do
    {
        Console.WriteLine("Numero: " + cont);
        cont++;
    } while (cont <= 10);
    Console.ReadKey();
}
```

### 6.5.3. For()

Diferente do While(), a instrução For() é capaz de definir, em uma única linha, a variável e o seu tipo, bem como estabelecer a condição para a estrutura e indicar o contador (figura 234).

**Figura 234**

Instrução for().

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura FOR");
    for(int cont=0; cont<=10; cont++)
    {
        Console.WriteLine("Numero: " + cont);
    }
    Console.ReadKey();
}
```

### 6.5.4. Break e continue

As instruções break e continue podem interferir diretamente em um laço de repetição. No próximo exemplo (figura 235), quando a variável "cont" tiver o valor "5", o loop será encerrado, executando a próxima instrução após o fechamento da estrutura.

**Figura 235**

Instruções break e continue.

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura BREAK");
    int num;
    for (num = 0; num < 10; num++)
    {
        if (num == 5){
            break;
        }
        Console.WriteLine ("O número é: " + num);
    }
    Console.WriteLine ("Mensagem após o laço");
    Console.ReadKey();
}
```

Observe, na figura 236 como fica o mesmo exemplo de quebra de loop com a instrução break, agora dentro da instrução While().

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura BREAK");
    int num=0;
    while (num < 1000) {
        num +=10;
        if (num > 100){
            break;}
        Console.WriteLine("O número é: " + num);
    }
    Console.WriteLine ("Mensagem após o laço");
    Console.ReadKey();
}
```

A instrução Continue força a execução do loop a partir do ponto em que está. No exemplo (figura 237), será forçada a contagem quando a variável "num" possuir o valor 100. Como resultado dessa operação, o valor 100 não será apresentado na tela.

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura CONTINUE");
    int num=0;
    while (num < 1000){
        num += 1;
        if (num == 100)
        {
            continue;
        }
        Console.WriteLine("O número é: " + num);
    }
    Console.WriteLine ("Mensagem após o laço");
    Console.ReadKey();
}
```

Na figura 238, confira outro exemplo da instrução continue, agora, dentro de uma estrutura for().

```
static void Main(string[] args)
{
    Console.WriteLine("Estrutura CONTINUE");
    for (int num = 0; num <= 10; ++num)
    {
        if (num == 5){
```

**Figura 236**

Exemplo com break dentro de While().

**Figura 237**

Exemplo de uso da instrução continue.

**Figura 238**

Exemplo de uso da instrução continue dentro de for().

```

        continue;
    }
    Console.WriteLine("O número é: " + num);
}
Console.WriteLine ("Mensagem após o laço");
Console.ReadKey();
}
    
```

### 6.6. Tratamento de erros / exceções

Uma das tarefas mais importantes dos programadores é saber realizar o tratamento de erros (consulte o quadro *Erros mais comuns*) e exceções que podem ocorrer durante a execução de um projeto. Seja por uma operação inválida ou até mesmo devido à abertura de determinado arquivo inexistente. Para isso, utilizamos uma estrutura bem familiar: o try-catch-finally (já abordado anteriormente, quando falamos de Java).

```

Try {
    // instrução que pode gerar o erro de execução
}
Catch
{
    // o que deve ser feito se ocorrer o erro
}
    
```

Outra possibilidade é incluir o finally, o que é opcional, pois ele sempre será executado, ocorrendo exceção ou não.

```

Try {
    // instrução que pode gerar o erro de execução
}
Catch
{
    // o que deve ser feito se ocorrer o erro
}
Finally
{
    // opcional, mas executado
}
    
```

O próximo exemplo (figura 239) mostra a conversão do conteúdo de dois TextBox, transformando os dados digitados em números do tipo float. Nesse ponto, pode ocorrer um erro, caso o usuário não digite os dados passíveis de conversão, como letra ou símbolos.

#### ERROS MAIS COMUNS

- Rede ou Internet:** geralmente por problemas de conexão (servidor, linha etc.).
- Drive:** falta da unidade de disco.
- Path:** caminho para a localização de arquivos em geral.
- Impressora:** equipamento não disponível, sem tinta ou sem papel.
- Componente não instalado:** falta algum componente de software ou está com erro de execução.
- Permissão:** privilégio para acesso de arquivos, dados ou área de rede.
- Clipboard:** problema com dados transferidos para determinada área.

```

try
{
    double var01 = double.Parse(txtLado01.Text);
    double var02 = double.Parse(txtLado02.Text);
    double resp = var01 * var02;
    txtResp.Text = resp.ToString();
}
catch
{
    MessageBox.Show("Dados Incorretos");
}
    
```

No caso da implementação do código com o finally, teríamos uma instrução ou um bloco de instruções sendo executados, independentemente da existência de uma exception. Ou seja, sempre será executado o bloco do finally, como é possível observar na figura 240.

```

try
{
    double var01 = double.Parse(txtLado01.Text);
    double var02 = double.Parse(txtLado02.Text);
    double resp = var01 * var02;
    txtResp.Text = resp.ToString();
}
catch
{
    MessageBox.Show("Dados Incorretos");
}
finally {
    MessageBox.Show("Mensagem de Finalização", "Mensagem");
}
    
```

#### 6.6.1. Exception

Quando o bloco catch{} é executado, poderá ser disparada uma exceção. Isso significa que foi gerado um código de erro e uma descrição correspondente. Cada erro possui características diferentes. Com bases nessas informações, podemos especificar o número do erro gerado.

```

catch (Exception objeto)
    
```

Ao modificar o exemplo anterior, serão emitidas mensagens de erro, como mostra a figura 241.

**Figura 239**  
Conversão do conteúdo de dois TextBox.

**Figura 240**  
Implementação do código com o finally.



**Figura 241**

Emissão de mensagens de erro.

```
try
{
    double var01 = double.Parse(txtLado01.Text);
    double var02 = double.Parse(txtLado02.Text);
    double resp = var01 * var02;
    txtResp.Text = resp.ToString();
}
catch (Exception erro) // erro é o objeto de controle
{
    MessageBox.Show("Dados Incorretos: Forneça apenas valores numéricos");
    MessageBox.Show(erro.Message);
    MessageBox.Show(erro.Source);
}
finally {
    MessageBox.Show("Mensagem de Finalização", "Mensagem");
}
```

### 6.7. Vetores e matrizes

Vamos ver agora, na figura 242, a forma de declaração, atribuição e acesso aos valores para diferentes tipos de vetores e matrizes. Uma boa dica, para essa etapa, é revisar os conceitos a respeito, no tema programação em Java.

**Figura 242**

Vetores e matrizes.

```
// vetor do tipo String
string[] j;
j = new string[2];
j[0] = "seg";
j[1] = "ter";
MessageBox.Show(j[0]);

string[] semana = { "dom", "seg", "ter", "qua", "qui", "sex" };
MessageBox.Show(semana[0]);

// vetor do tipo float
float[] y;
y = new float[3];
y[0] = 10.5F;
y[1] = 7.3F;
y[2] = 1.9F;

// vetor do tipo inteiro
int[] x = { 10, 5, 3 };
MessageBox.Show(x[0].ToString());
```

```
// matriz do tipo double
double[,] matriz;
matriz = new double[2,2];
matriz[0,0] = 1;
matriz[0,1] = 2;
matriz[1,0] = 3;
matriz[1,1] = 4;
MessageBox.Show(matriz[1,1].ToString());

// matriz do tipo inteiro
int[,] temp = { { 1, 4 }, { 2, 7 }, { 3, 5 } };
MessageBox.Show(temp[1, 1].ToString());
```

### 6.8. Classes

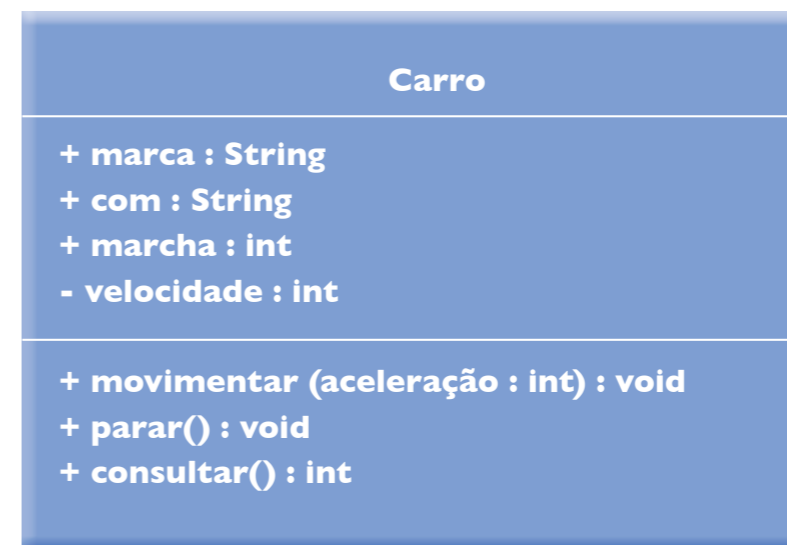
Usando os mesmos conceitos do Java, o C# pode implementar classes específicas para programação, as quais seguem os mesmos princípios de formação e manipulação. Para relembrar os conceitos de programação orientados a objetos, observe o exemplo a seguir e, depois, confira a classe na figura 243.

Uma classe chamada Carro tem atributos referentes ligados a marca, combustível e marcha e, ainda, os métodos movimentar(), parar() e consultar(), que representam:

- **Movimentar()**: indica a velocidade com que o veículo deve andar.
- **Parar()**: quando acionado, diminui em uma unidade a velocidade do veículo.
- **Consultar()**: informa a atual velocidade do veículo.

**Figura 243**

Diagrama de classe.



Observe, então, na figura 244, como ficaria a classe, seguindo o diagrama.

**Figura 244**

Como ficou a classe, de acordo com o diagrama.

```
class Carro
{
    // atributos da classe carro
    public string marca;
    public string comb;
    public int marcha;
    private int velocidade;

    // método movimentar
    // recebe o valor (int) que indica a aceleração do veículo
    public void movimentar(int aceleracao){
        velocidade += aceleracao;
    }

    // método parar
    // diminui em uma unidade toda vez que é acionado
    public void parar(){
        if (velocidade > 0){
            velocidade--;
        }
    }

    // método consultar
    // retorna a velocidade do veículo
    public int consultar(){
        return velocidade;
    }
}
```

Analisando o código principal main(), na figura 245, veja o que aparece.

**Figura 245**

Analisando o código main().

```
static void Main(string[] args)
{
    // instanciando a classe Carro
    Carro objCarro = new Carro();

    // atribui os valores para o OBJ
    objCarro.marca = "Fusca";
    objCarro.comb = "Querozene";
    objCarro.marcha = 4;
}
```

```
// imprime os dados do veículo e combustível
Console.WriteLine("Veículo: " + objCarro.marca + " de " + objCarro.marca + " marchas");
Console.WriteLine("Combustível: " + objCarro.comb + "\n");

// inicia a movimentação do carro com velocidade 5
objCarro.movimentar(5);

// consulta a velocidade do veículo
velocidade = objCarro.consultar();

// visualiza a velocidade do veículo
Console.WriteLine("Velocidade Atual:" + velocidade);
Console.ReadKey();

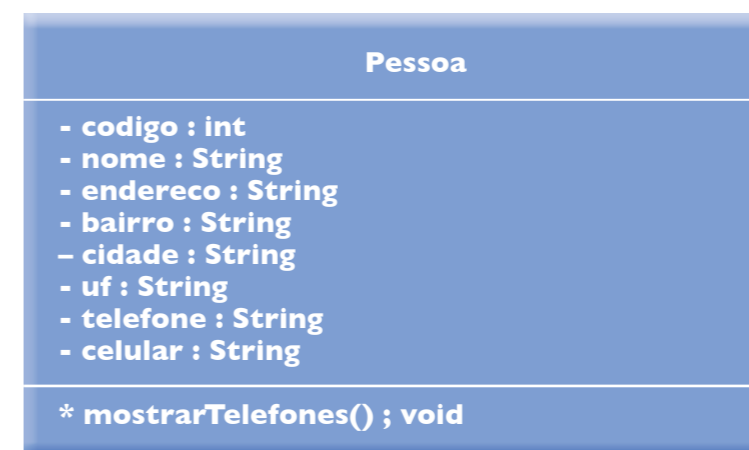
// aumenta a velocidade do veículo
objCarro.movimentar(5);
Console.WriteLine("Velocidade Atual:" + objCarro.consultar());
Console.ReadKey();

// começa a parar o carro
// a primeira chamada diminui em 1 unidade a velocidade
objCarro.parar();
velocidade = objCarro.consultar();
Console.WriteLine("Velocidade Atual:" + objCarro.consultar());
Console.ReadKey();
}
```

Assim como em Java, é importante seguir as boas práticas de programação. Uma delas é utilizar os getters e setters. Vamos usar uma classe criada no capítulo anterior para compor outro exemplo (figuras 246 e 247).

**Figura 246**

Classe Pessoa.



**Figura 247**

Uso de getters e setters.

```
public class Pessoa
{
    private int codigo;
    private string nome;
    private string endereco;
    private string bairro;
    private string cidade;
    private string uf;
    private string telefone;
    private string celular;

    public int setCodigo(int novoCodigo){
        this.codigo = novoCodigo;
    }
    public int getCodigo(){
        return this.codigo;
    }
    public String setNome(String novoNome){
        this.nome = novoNome;
    }
    public int getNome(){
        return this.nome;
    }
    // o get e set deverá ser construído para os demais atributos
}
```

### 6.9. Windows Form Application – componentes

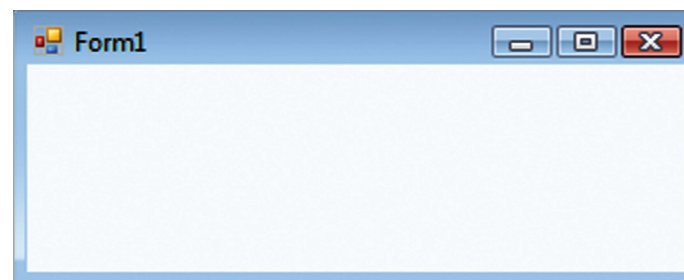
O tipo Windows Form Application (em português aplicação de formulário Windows), diferentemente do Console Application, permite a elaboração do projeto para ambiente Windows, utilizando componentes fornecidos pela Toolbox (caixa de ferramentas). Essa janela apresenta uma série de componentes e, cada um deles, uma variedade de propriedades que podem ser configuradas tanto via janela, como por meio de programação. Veja, a seguir, os componentes mais comuns para a elaboração de um projeto.

#### 6.9.1. Form

Ao iniciar a aplicação do Form (formulário), será disponibilizado o Form1 (nome padrão) para o desenvolvimento dos trabalhos. Esse é o repositório principal para os nossos componentes (figura 248). Confira os detalhes no quadro *Propriedades Form*.

**Figura 248**

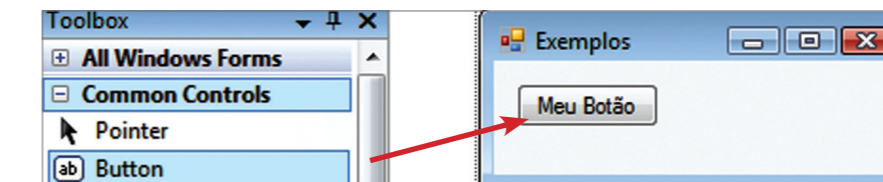
Form.



PROPRIEDADES FORM	
Name	Modifica o nome
Text	Atualiza a barra de título do formulário
BackColor	Cor de fundo do formulário
BackgroundImage	Insere uma imagem como plano de fundo em um formulário
BackgroundImageLayout	Ajusta a posição da imagem em relação ao formulário
ControlBox	Desativa os botões maximizar, minimizar e fechar
FormBorderStyle	Muda as configurações de visualização do formulário
Icon	Insere um ícone no formulário
MaximizeBox	Permite ou não ao usuário maximizar
MinimizeBox	Permite ou não ao usuário minimizar
Size	Define a largura e a altura do formulário
WindowState	Define o modo como o formulário será aberto: maximizado, minimizado, etc.

#### 6.9.2. Button

O Button (botão), apresentado na figura 249, é o responsável por grande parte da nossa programação. Ao clicar sobre esse componente, acessamos a janela de códigos, na qual o primeiro evento está previamente selecionado, nesse caso “click”. Isso indica que o código será executado quando dermos o clique sobre o botão (figura 250). Confira detalhes no quadro *Propriedades Button*.



**Figura 249**

Button.

**Nome do botão e evento**

```
private void btnMeuBotao_Click(object sender, EventArgs e)
{
    // Área de programação
}
```

**Área de programação**

**Figura 250**

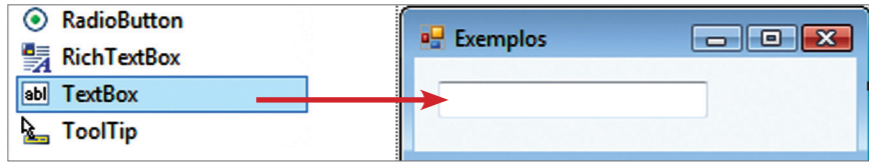
Evento Click.

PROPRIEDADES BUTTON	
Name	Modifica o nome
Text	Texto para o botão
BackColor	Modifica a cor do botão
BackgroundImage	Insere uma imagem como plano de fundo
BackgroundImageLayout	Ajusta a posição da imagem em relação ao botão
Visible	Define se esse botão está visível ou não

### 6.9.3. TextBox

O TextBox (caixa de texto) é o componente responsável por receber as informações do usuário (figura 251), é também o item mais comum, pois a maioria das entradas de dados é realizada por ele. Observe detalhes no quadro *Propriedades TextBox*.

**Figura 251**  
TextBox.

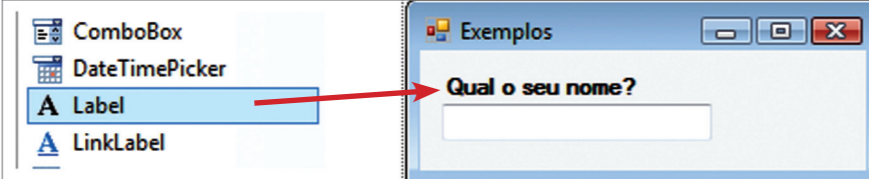


PROPRIEDADES TEXTBOX	
Name	Modifica o nome
Text	Insere um texto predefinido
BackColor	Cor de fundo da caixa de entrada
ForeColor	Cor da fonte
CharacterCasing	Controla a entrada do texto, mantendo as letras em maiúscula ou minúscula
MaxLength	Tamanho máximo em número de caracteres
PasswordChar	Caractere utilizado para coletar senha
ReadOnly	Mostra o texto, mas não permite que ele seja alterado
TextAlign	Define se o texto deve ser colocado à direita, à esquerda ou centralizado

### 6.9.4. Label

Usamos o Label (rótulo) para inserir rótulos nos formulários, como mostra a figura 252 (consulte o quadro *Propriedades Label*, para obter detalhes).

**Figura 252**  
Label.

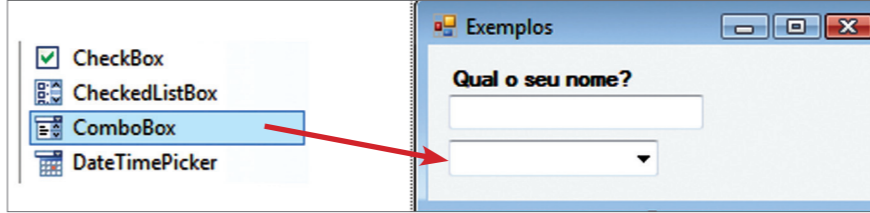


PROPRIEDADES LABEL	
Name	Modifica o nome
Text	Insere um texto predefinido
BackColor	Cor de fundo da caixa de entrada, local onde as informações serão inseridas
ForeColor	Cor da fonte
Font	Define a fonte do texto

### 6.9.5. ComboBox

O ComboBox (caixa de agrupamento) permite ao usuário abrir várias opções (figura 253), assim como ocorre quando escolhemos uma fonte de letra do Microsoft Word. Veja mais detalhes no quadro *Propriedades ComboBox*, os detalhes.

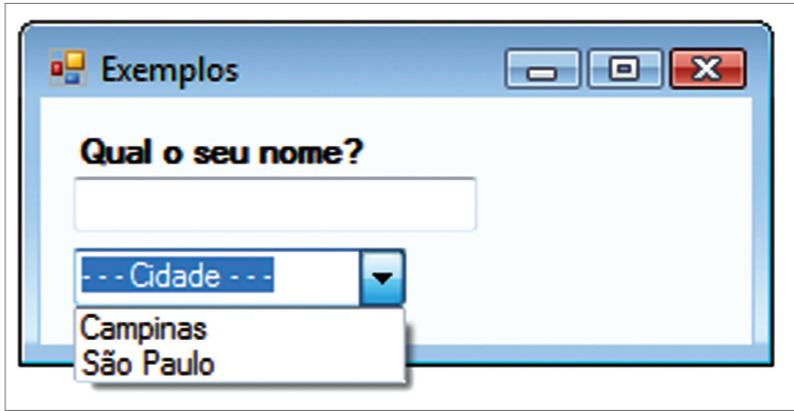
**Figura 253**  
ComboBox.



PROPRIEDADES COMBOBOX	
Name	Modifica o nome
Text	Insere um texto no combo
DataSource	Pode ser ligado a uma base de dados DataTable
Items	Lista de valores que o ComboBox disponibiliza ao usuário para seleção

Para a inserção de itens, escolha a opção *Items*. Uma nova caixa de diálogo será aberta: os itens deverão ser colocados um abaixo do outro (one per line). Após a confirmação, teremos o ComboBox carregado com as informações (figura 254).

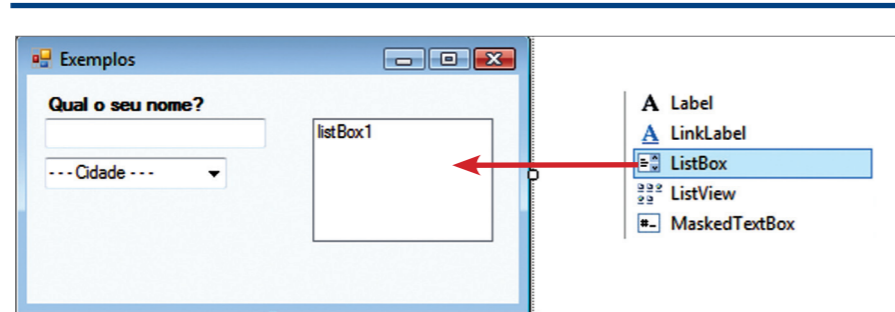
**Figura 254**  
ComboBox (carregado).



### 6.9.6. ListBox

Diferentemente do ComboBox, o ListBox (caixa de lista) disponibiliza várias opções aos usuários, porém, em forma de lista. Isso permite a utilização de barra de rolagem caso o número de opções ultrapasse o limite da caixa da janela (figura 255). Consulte o quadro *Propriedades ListBox*.

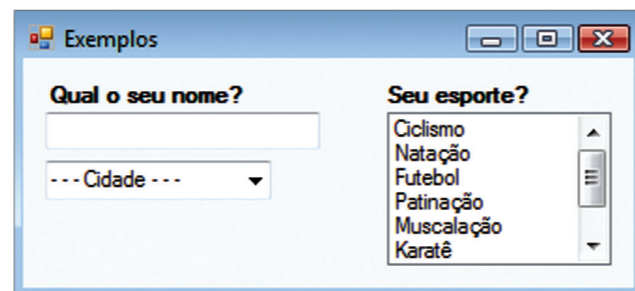
**Figura 255**  
ListBox.



PROPRIEDADES LISTBOX	
Name	Modifica o nome
DataSource	Pode ser ligado a uma base de dados DataTable
Items	Lista de valores que o ComboBox disponibiliza ao usuário para seleção
SelectionMode	Permite escolher um ou mais itens de uma só vez

Para carregar o ListBox (figura 256), use o mesmo procedimento do ComboBox.

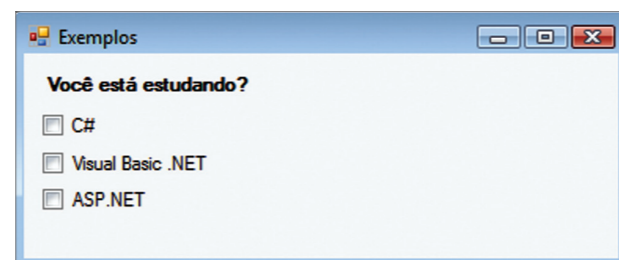
**Figura 256**  
ListBox (carregado).



### 6.9.7. CheckBox

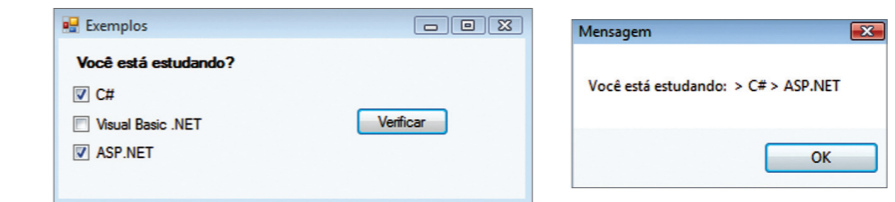
Utilizamos o controle CheckBox (caixa de seleção) para obter múltiplas opções de resposta ou para simular um “sim” ou “não”, dependendo do escopo empregado. O exemplo da figura 257 simula um questionário no qual o usuário deve marcar quais linguagens de programação está estudando (consulte também o quadro *Propriedades CheckBox*).

**Figura 257**  
CheckBox.



PROPRIEDADES CHECKBOX	
Name	Modifica o nome
Text	Insere a texto da opção
CheckState	Deixa a opção já selecionada

Para verificar o que foi selecionado pelo usuário, devemos realizar o teste em cada um dos elementos. Assim, implementaremos o código anterior com um botão, para verificação dos itens selecionados, cuja saída será uma caixa de diálogo contendo todas as informações selecionadas (figuras 258 a e 258 b).



**Figuras 258 a e 258 b**  
Verificação da caixa CheckBox.

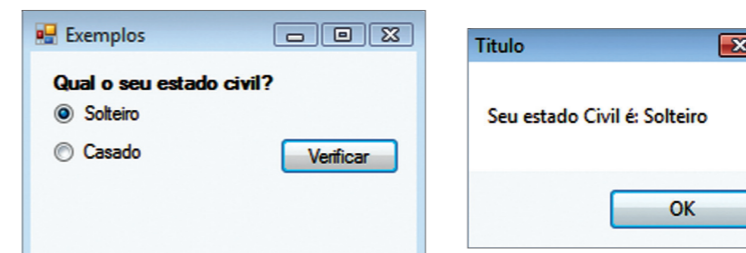
O código descrito refere-se apenas ao evento click, relacionado ao botão de verificação (confira na figura 259).

```
private void btnVerificar_Click(object sender, EventArgs e)
{
    string frase = "Você está estudando: ";
    if (chkOpcao1.Checked == true)
        frase = frase + "> C#";
    if (chkOpcao2.Checked == true)
        frase = frase + "> Visual Basic .NET";
    if (chkOpcao3.Checked == true)
        frase = frase + "> ASP.NET";
    MessageBox.Show(frase, "Mensagem");
}
```

**Figura 259**  
Código descrito do evento click, com a verificação.

### 6.9.8. RadioButton

O RadioButton (botão de seleção) é diferente do CheckBox, pois pode estabelecer relações entre si, o que possibilita fornecer múltiplas opções para que se escolha somente uma. O exemplo ilustrado nas figuras 260 (a e b) e 261 verifica o estado civil do usuário (confira também o quadro *Propriedades RadioButton*).



**Figuras 260**  
Verificação da opção do RadioButton.

**Figura 261**  
O detalhamento da verificação.

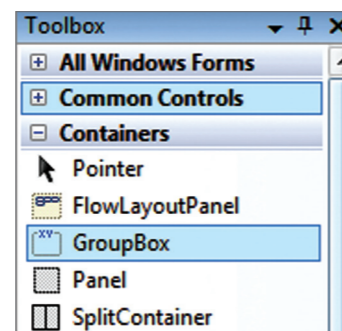
```
private void btnVerificar_Click(object sender, EventArgs e)
{
    string frase = "Seu estado Civil é:";
    if (rdbCasado.Checked == true)
        frase = frase + " Casado";
    if (rdbSolteiro.Checked == true)
        frase = frase + " Solteiro";
    MessageBox.Show(frase, "Titulo");
}
```

PROPRIEDADES RADIOBUTTON	
Name	Modifica o nome
Text	Insera a texto da opção
CheckState	Deixa a opção já selecionada

### 6.9.8.1. Agrupamento

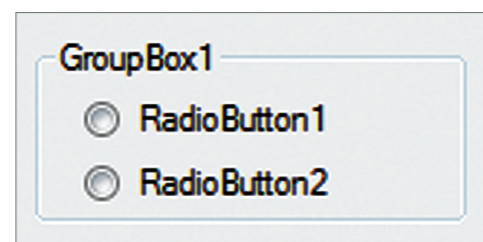
Em alguns momentos, será necessário reunir grupos de opção usando o RadioButton. Para que os controles fiquem vinculados, mas dentro de um determinado grupo, devemos utilizar um container, ou melhor, uma estrutura que permita criar tal vínculo. Selecione, então, o componente groupBox da aba Containers da janela Toolbox (figura 262). Confira, também, o quadro *Propriedades Agrupamento*.

**Figura 262**  
Container – groupBox.



Coloque o container no formulário e, em seguida, o RadioButton dentro do container, mas sem arrastar, apenas inserindo dentro do grupo (figura 263).

**Figura 263**  
RadioButton e groupBox.

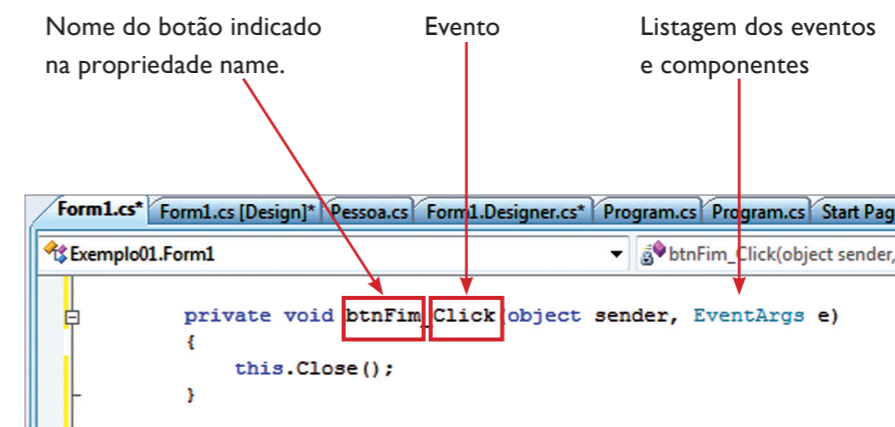


PROPRIEDADES AGRUPAMENTO	
Name	Modifica o nome
Text	Título da caixa de grupo

## 6.10. Eventos

Para cada componente inserido na aplicação, incluindo o formulário, podemos manipular eventos distintos. O evento é a forma com que a classe se manifesta quando o usuário interage com os componentes do formulário: com um clique, duplo clique, passagem de mouse etc. Por exemplo, ao inserir um botão para finalizar a aplicação, devemos associar o código a um determinado evento, no caso, o clique. Na figura 264, podemos verificar a identificação do botão de acordo com a atribuição do seu name e com o evento que receberá a programação.

**Figura 264**  
Eventos.



Para modificar os eventos, podemos utilizar a janela de Properties (figura 265) com o botão Events, semelhante a um raio.

**Figura 265**  
Para modificação de eventos.

