ESCOLA
Alcides Maya

Linguagem de Programação I (linguagem C#)

1 INTRODUÇÃO

Em Junho de 2000 a Microsoft anunciou a Plataforma .NET e uma nova linguagem de programação chamada C# (se lê "C Sharp"). C# é uma linguagem fortemente tipada e orientada a objetos projetada para oferecer a melhor combinação de simplicidade, expressividade e performance.

A linguagem C# aproveita conceitos de muitas outras linguagens, mas especialmente de C++ e Java. Ela foi criada por Anders Hejlsberg (que já era famoso por ter criado o TurboPascal e o Delphi para a Borland) e Scott Wiltamuth.

A Plataforma .NET é centrada ao redor de uma Common Language Runtime (CLR, conceito similar ao da Java Virtual Machine, JVM) e um conjunto de bibliotecas que pode ser empregado em uma grande variedade de linguagens, as quais podem trabalhar juntas, já que todas são compiladas para uma mesma linguagem intermediária, a Microsoft Internediate Language (MSIL). Assim, é possível desenvolver aplicativos mesclando C# e Visual Basic ou qualquer outra linguagem suportada.

A sintaxe utilizada pelo C# é relativamente fácil, o que diminui o tempo de aprendizado. Todos os programas desenvolvidos devem ser compilados, gerando um arquivo com a extensão DLL ou EXE. Isso torna a execução dos programas mais rápida se comparados com as linguagens de script (VBScript , JavaScript) que atualmente utilizamos na internet.

Poderíamos citar entre algumas das principais características do C#:

- Orientada a Objetos
- Não há variáveis ou funções globais. Todos os métodos e atributos devem ser declarados dentro de classes. Atributos e métodos estáticos de classes públicas podem servir como substitutos para variáveis e métodos globais.
- Apontadores
- Em C#, apontadores só podem ser usados dentro de blocos especificamente marcados como inseguros. Programas com código inseguro precisam das permissões apropriadas para serem executados.
- Memória Gerenciada
- Em C# a memória não precisa ser explicitamente liberada. Ao invés disso ela é automaticamente gerenciada por meio do Garbage Collector (coletor de lixo), que constantemente percore a memória alocada para verificar se ela ainda é necessária, eliminando assim o risco de vazamentos de memória.
- Tipagem Forte
- C# é mais fortemente tipada do que C++: as única conversões implícitas por default são aquelas que são consideradas seguras, como por exemplo o armazenamento de um inteiro em um tipo de dados maior ou a conversão de um tipo derivado para um tipo base. Não há conversão implícita entre booleanos e inteiros.

2 O AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento do C#, Microsoft Visual Studio, implementa o conceito de RAD (Rapid Application Development), oferecendo ferramentas para a construção rápida de aplicativos, ao mesmo tempo que oferece um excelente designer de telas e debugger integrado.

Entre os diversos recursos do ambiente podemos destacar:

- O **Editor de Código** (Code Editor), usado para manipular o código fonte;
- O **Compilador C#** (C# Compiler), utilizado para converter o código fonte em um programa executável;
- O **Depurador do Visual Studio** (Visual Studio Debugger), usado para testar seus programas;
- A **Caixa de Ferramentas** (Toolbox) e o Editor de Formulários (Windows Forms Designer), para a rápida criação de interfaces com o usuário usando o mouse;
- O **Explorador de Soluções** (Solution Explorer), útil para o gerenciamento de arquivos de projeto e configurações;
- O **Editor de Projetos** (Project Designer), usado para configurar o compilador, caminhos de instalação e demais recursos;
- O **Visualizador de Classes** (Class View), usado para navegar através das classes definidas no seu código fonte;
- A Janela de Propriedades (Properties Window), utilizada para definir as propriedades e eventos nos controles da sua interface com o usuário;
- O **Navegador de Objetos** (Object Browser), que pode ser usado para ver os métodos e classes disponíveis em bibliotecas externas (arquivos DLL, por exemplo), inclusive os objetos do Framework .NET;
- O **Explorador de Documentos** (Document Explorer), que navega através da documentação do produto em seu computador local ou na Internet.

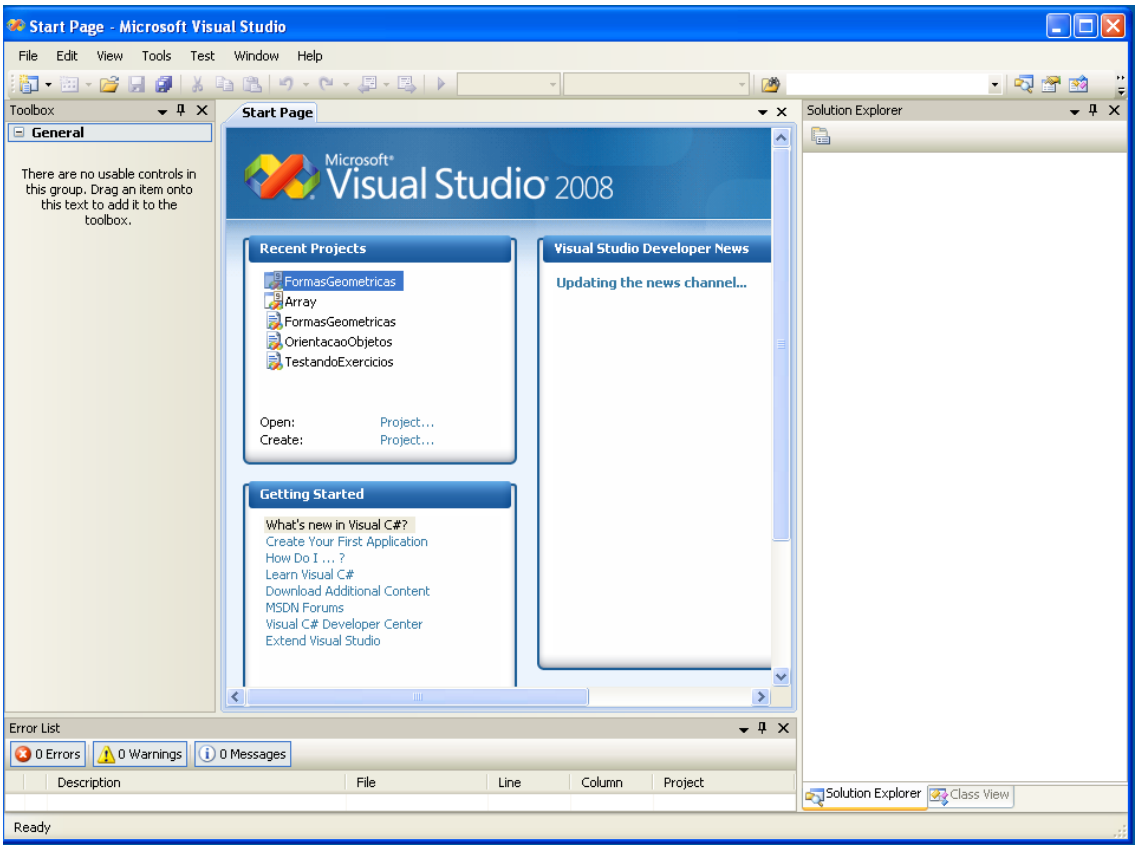
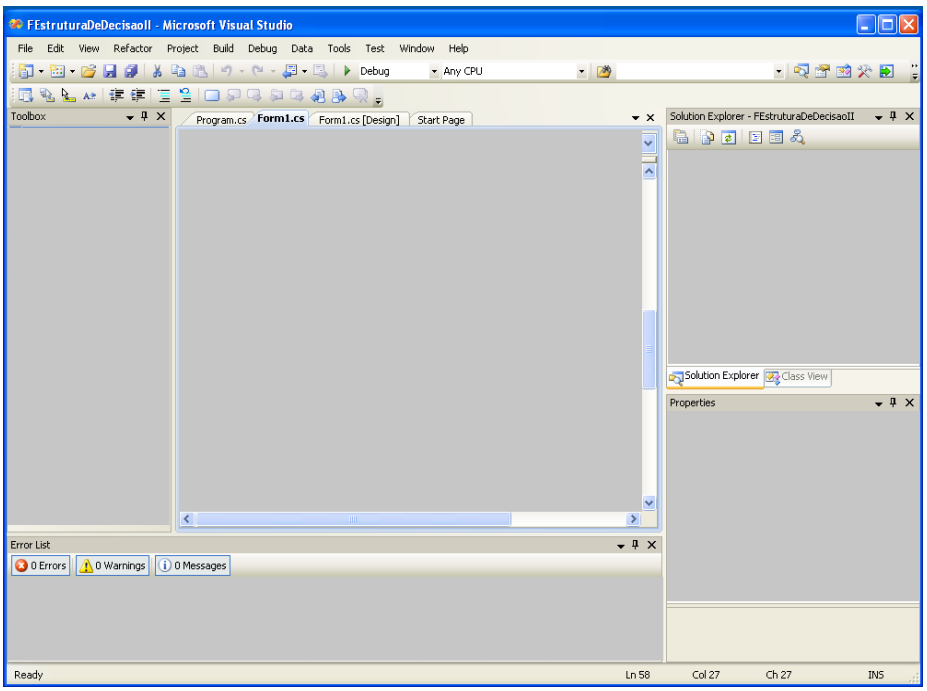


Ilustração 1- Tela Inicial do Visual Studio 2008



Ambiente de Desenvolvimento – IDE Visual Studio 2008.

3 CRIANDO UMA APLICAÇÃO NO VISUAL STUDIO 2008

No ambiente Visual Studio tomamos por base o conceito de Projetos. Cada Projeto pode ser baseado em qualquer um dos tipos de aplicações disponíveis, como por exemplo: WindowsFormsAplication, ConsoleAplication, ClassLibrary e várias outras.

Quando você clicar nas opções File > New Project... irá surgir uma nova janela onde você poderá escolher o tipo de aplicação a ser criada:

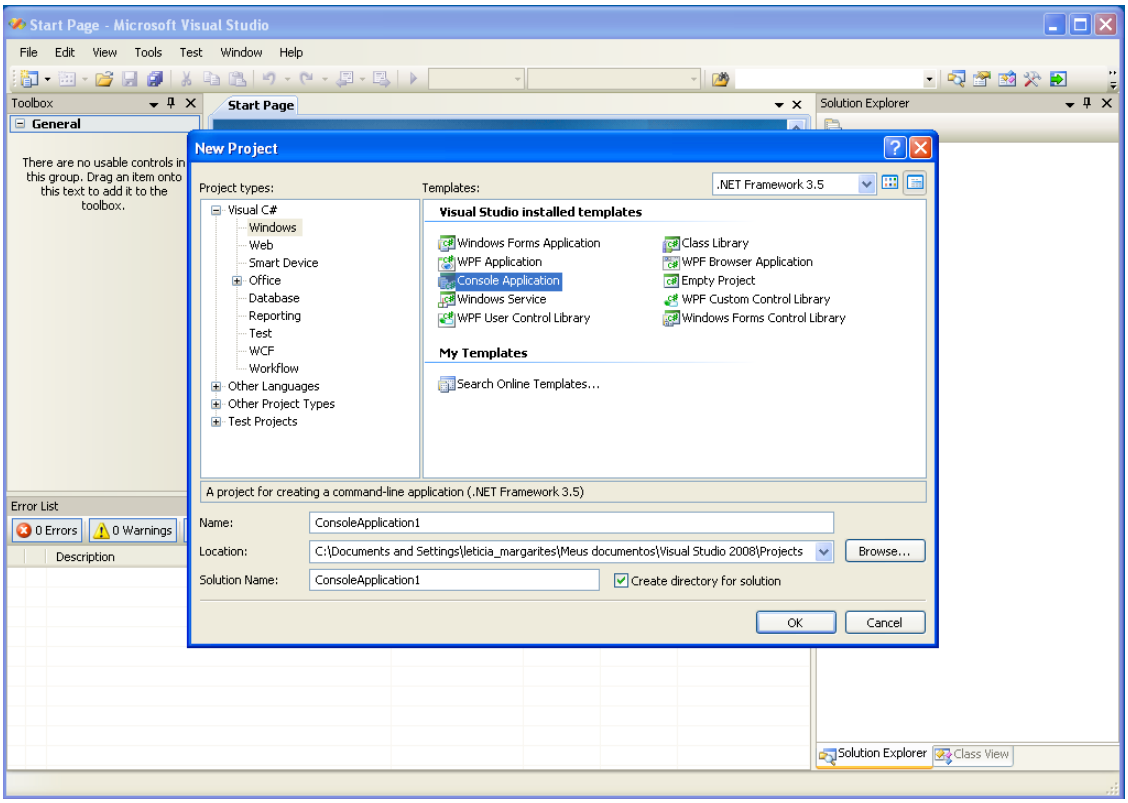


Ilustração 2 - Tela Incio de um novo Projeto

4 O SEU PRIMEIRO PROGRAMA EM C#

Nosso primeiro programa em C# é extremamente simples. O programa deve exibir na tela a mensagem “Olá Mundo !”.

Para começar, clique em File > New Project... e selecione o tipo Console Application. Na mesma caixa de diálogos, altere o nome sugerido para “OlaMundo” e clique em OK.

A seguir, use o editor de textos para alterar o código do programa criado automaticamente para você e adicione as linhas mostradas em negrito abaixo:

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

namespace OlaMundo
{
    class Program
    {
      static void Main(string[] args)
      {
                System.Console.WriteLine(“Olá Mundo!”);
                System.Console.ReadKey;
      }
    }
```

Uma vez completada a digitação, clique em File > Save All (ou pressione Ctrl-Shift-S) para salvar o seu projeto e todos os arquivos criados.

Depois, para compilar o programa, clique em Build > Build Solution (ou pressione F6). Por fim, para executá-lo, clique em Debug > Start Without Debugging (ou pressione Ctrl-F5). Pronto, você verá a mensagem escrita na tela. Pressione qualquer tecla para concluir o programa.

5 O BÁSICO SOBRE O C#

Todas as instruções devem estas entre chaves e sempre ser finalizadas com um ponto-e-vírgula, como você verá a seguir:

```
{
    // Código.
}
```

5.1 COMENTÁRIOS

Usamos comentários para descrever partes complexas de código, a fim de facilitar a manutenção, seja para quem elaborou o software, seja para terceiros. Comentários não são interpretados pelo compilador. Em C# existem dois tipos de comentários:

Barras duplas: convertem o restante da linha em comentários.

```
// Nosso primeiro programa em C#.
```

Blocos de Texto: os caracteres /* e */ definem blocos de textos como comentários.

```
/* Este é o meu primeiro contato com C#.
   Espero que aprenda rápido está nova linguagem.
   Obrigado. */
```

5.2 MÉTODO MAIN

Um programa C# deve conter um método especial chamado Main. Este é o primeiro método chamado quando se executa uma aplicação em C#, e é a partir dele que você criará objetos e executará outros métodos.

```
static void Main()
{
    // Código.
}
```

Ou retornar um valor inteiro (int):

```
static int Main()
{
    // Código.
    Return 0;
}
```

O método Main também pode receber parâmetros, por meio de um array de strings.

```
static void Main(string[] args)
{
    // Código.
}
```

Ou

```
static int Main(string[] args)
{
    // Código.
    Return 0;
}
```

5.3 ENTRADA E SAÍDA BÁSICA

A entrada e saída de um programa C# é realizado pela biblioteca de classes do .NET Framework. Entre as classes presentes nesta biblioteca podemos citar a classe System.Console, que pode ser usada para escrever na tela e obter dados do teclado.

5.3.1 IMPRIMINDO NA TELA

Imprimir na Tela em C# é uma atividade bastante simples. Para imprimir basta chamar o método System.Console. WriteLine(). Veja o exemplo:

```
System.Console.WriteLine("Olá, Mundo!");
```

Esta linha imprime a mensagem mostrada abaixo:

```
Olá, Mundo!
```

5.3.2 LENDO DO TECLADO

Ler dados a partir do teclado pode ser um pouco mais complicado, pois há mais opções e por vezes vamos precisar nos preocupar com as conversões de dados (veja mais sobre isso mais adiante). Mas por enquanto vamos nos ater ao básico.

5.3.3 ESPERANDO POR UMA TECLA

Por vezes você quer apenas que o usuário pressione uma tecla qualquer antes que o programa proceda com uma dada tarefa. Isso é simples de fazer, conforme vemos abaixo:

```
System.Console.ReadKey();
```

O método acima retorna informações sobre a tecla pressionada, mas por enquanto vamos ignorar isto, pois por ora não nos interessa saber qual foi. Veremos mais sobre isso mais tarde.

5.3.4 LENDO DADOS A PARTIR DO TECLADO

Em outras situações iremos querer que o usuário nos dê alguma informação. A maneira mais fácil de se fazer isso é ler uma linha a partir do teclado, como mostra o exemplo:

```
string nome = System.Console.ReadLine();
```

O que quer que o usuário digitasse seria armazenado na variável chamada nome (veja mais sobre variáveis na próxima seção).

Por vezes será necessário converter o texto que o usuário digitou para um outro tipo de dados, então é aí que entram as conversões de tipos de dados. Veja mais sobre tipos de dados e conversões na próxima seção.

6 VARIÁVEIS E TIPOS DE DADOS

6.1 VARIÁVEIS

As variáveis são utilizadas para armazenar informações na memória do computador enquanto o programa C# esta sendo executado. As informações contidas nas variáveis podem ser alteradas durante a execução do programa.

As variáveis devem possuir um nome para que possamos nos referenciar a elas mais tarde. Ao nomear uma variável devemos observar as seguintes restrições:

- O nome deve começar com uma letra ou _ .
- Não são permitidos espaços, pontos ou outros caracteres de pontuação, mas podemos usar números.
- O nome não pode ser uma palavra reservada do C#.
- O nome deve ser único dentro do contexto atual.

Além do nome, devemos também definir o tipo de dados e o escopo (local onde a variável estará acessível). O escopo é definido pelos modificadores de acesso (veremos mais sobre isso mais tarde)

Para declarar uma variável, primeiro você precisa indicar o seu tipo e depois o seu nome. Veja os exemplos:

```
string nome;
int telefone;
```

6.2 TIPOS DE DADOS

Como toda a linguagem de programação o C# apresenta seu grupo de tipos de dados básico. Esses tipos são conhecidos como tipos primitivos ou fundamentais por serem suportados diretamente pelo compilador, e serão utilizados durante a codificação na definição de variáveis, parâmetros, declarações e até mesmo em comparações.

Em C# todo o tipo de dados possui um correspondente na CLR (Common Language Runtime), por exemplo: int em C# refere-se a System.Int32 na plataforma .NET.

Tipo C#	Tipo .NET	Descrição	Faixa de dados
bool	System.Boolean	Booleano	true ou false
byte	System.Byte	Inteiro de 8-bit com sinal	-127 a 128
char	System.Char	Caracter Unicode de 16-bit	U+0000 a U+ffff
decimal	System.Decimal	Inteiro de 96-bit com sinal com 28-29 dígitos	1,0 × 10-28 a 7,9 × 1028
double	System.Double	Flutuante IEEE 64-bit com 15-16 dígitos significativos	±5,0 × 10-324 a ±1,7 × 10308
float	System.Single	Flutuante IEEE 32-bit com 7 dígitos significativos	±1,5 × 10-45 a ±3,4 × 1038
int	System.Int32	Inteiro de 32-bit com sinal	-2.147.483.648 a 2.147.483.647
long	System.Int64	Inteiro de 64-bit com sinal	–9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
Object	System.Object	Classe base	
Sbyte	System.Sbyte	Inteiro de 8-bit sem sinal	0 a 255
Short	System.Int16	Inteiro de 16-bit com sinal	-32,768 a 32,767
String	System.String	String de caracteres Unicode	
Uint	System.UInt32	Inteiro de 32-bit sem sinal	0 a 4,294,967,295
Ulong	System.UInt64	Inteiro de 64-bit sem sinal	0 a 18,446,744,073,709,551,615
Ushort	System.UInt16	Inteiro de 16-bit sem sinal	0 a 65,535

6.3 CONVERSÕES DE TIPOS

Uma tarefa muito comum no mundo da programação é a utilização de conversões. Por exemplo: podemos converter um inteiro para um longo. Quando o usuário digita um número qualquer numa TextBox, é necessária uma conversão de valores antes que um cálculo possa ser realizado.

Método	Descrição
System,Convert.ToBoolean()	Converte uma string para um valor booleano
System,Convert.ToByte()	Converte para o tipo byte
System,Convert.ToChar()	Converte para o tipo char
System,Convert.ToDateTime()	Converte para o tipo data e hora
System,Convert.ToDecimal()	Converte para o tipo decimal
System,Convert.ToDouble()	Converte para o tipo doublé
System,Convert.ToInt16()	Converte para o tipo short
System,Convert.ToInt32()	Converte para o tipo inteiro
System,Convert.ToInt64()	Converte para o tipo long
System,Convert.ToSingle()	Converte para o tipo single
System,Convert.ToString()	Converte para o tipo string

Alguns tipos são convertidos automaticamente se o valor que receber a conversão puder conter todos os valores da expressão. A tabela a seguir mostra as conversões que são realizadas automaticamente.

De	Para
sbyte	short, int, long, float, double ou decimal
byte	short, ushort, int, uint, long, float, double ou decimal
short	int, long, float, double ou decimal
ushort	int, uint, long, ulong, float, double ou decimal
int	long, float, double ou decimal
uint	long, ulong, float, double ou decimal
long	float, double ou decimal
ulong	float, double ou decimal
char	ushort, int, uint, long, ulong, float, double ou decimal
float	double

7 ARRAYS

Array é uma coleção de elementos armazenados em seqüência, acessíveis através de um índice numérico. No C#, o primeiro elemento de um array é o de índice zero (0).

É importante notar que podemos criar arrays com uma ou mais dimensões.

Para definir um array você seguirá esta estrutura:

```
<tipo-de-dados> [] <nome-do-array> = new <tipo-de-dados>[<tamanho>];
```

7.1 ARRAYS DE DIMENSÃO ÚNICA

Você pode criar um array e não inicializá-lo:

```
string[] arr;
```

No entanto, para utilizá-lo em outras partes do código, precisamos inicializá-lo. Por exemplo, para inicializarmos o array anterior com 10 elementos:

```
arr = new string[10];
```

Para armazenar informações num array use o índice para indicar o elemento desejado:

```
arr[0] = “Alfredo”;
arr[1] = “Maria”;
arr[2] = “Paulo”;
arr[8] = “Beatriz”;
```

Entretanto, podemos inicializá-lo, junto com a declaração:

```
arr = new string[4] {“Alfredo”, “Maria”, “Paulo”, Beatriz”};
```

Podemos também omitir o número de elementos:

```
int [] numeros = {1, 2, 3, 4, 5};
string [] nomes = {“Alfredo”, “Maria”, “Paulo”, “Beatriz”};
```

Para acessar um elemento de um array você deve usar o índice do elemento desejado, lembrando que os índices começam sempre eo zero. Por exemplo, para acessar o segundo elemento do array nomes você usaria a seguinte instrução:

```
string saida = arr [1];   // Isto armazenaria “Maria” na
                                    // variável saida
```

7.2 ARRAYS MULTIDIMENSIONAIS

Num array multidimensional separamos as dimensões por vírgulas.

Para declará-lo com duas dimensões, use:

```
int [ , ] arr;
```

Para declará-lo com três dimensões, use:

```
int [ , , ] arr;
```

Exemplos:

```
int [,] numeros = new int [3, 2] {{1, 2}, {3, 4}, {5, 6} };
string [,] nomes = new string [2, 2] {{“Mara”, “Lotar”}, {“Mary”, “José”} };
```

Essas linhas poderiam ser representadas por:

```
numeros [0, 0] = 1;
numeros [0, 1] = 2;
numeros [1, 0] = 3;
numeros [1, 1] = 4;
numeros [2, 0] = 5;
numeros [2, 1] = 6;
```

E por:

```
nomes [0, 0] = “Mara”;
nomes [0, 1] = “Mary”;
nomes [1, 0] = “Lotar”;
nomes [1, 1] = “José”;
```

8 EXPRESSÕES

Um programa não será muito útil se não pudermos usá-lo para efetuar cálculos ou outro tipo de manipulações de dados. É para isso que surgem os operadores.

Conforme veremos a seguir, existem diferentes tipos de operadores, dependendo do tipo de operação que desejamos realizar.

Categoria	Operadores
Aritmética	+ - * / %
Lógica (boolena e bitwise)	& | ^ ! ~ && || true false
Concatenação de string	+
Incremento e decremento	++ --
Shift	<< >>
Relacional	== != < > <= >=
Atribuição	= += -= *= /= %= &= |= ^= <<= >>=
Acesso a membro	.
Indexação	[]
Cast	()
Condicional	?:
Delegate (concatenação e remoção)	+ -
Criação de objeto	new
Informação de tipo	is sizeof typeof
Controle de excessão de overflow	checked unchecked
Indireção e endereço	* -> [] &

9 ESTRUTURAS DE CONTROLE

Uma instrução de controle é usada para controlar a execução de partes de um programa. Sem as instruções de controle um programa seria executado da primeira à última linha. Isso pode ser algo que não queremos em situações nas quais uma instrução deve ser executada somente se determina condição for verdadeira. As instruções de controle utilizadas no C# são if, switch e else if.

9.1 ESTRUTURAS DE DECISÃO

INSTRUÇÃO IF

A instrução if pode ser usada para seletivamente executar trechos de código. Você pode executar um códgo apenas se uma dada condição for verdadeira ou falsa. Veja o exemplo:

```
if (<condição>)
{
    // Código para quando a condição for verdadeira.
}
```

Existe ainda a possibilidade de se usar a cláusula opcional else, que inverte a condição testada. Seu corpo será executado se a condição não for verdadeira.

```
if (<condição>)
{
    // Código para quando a condição for verdadeira.
}
else
{
    // Código para quando a condição for falsa.
}
```

Se a condição for satisfeita, todas as instruções do primeiro bloco de comando serão executadas; no entanto, se a condição não for satisfeita, serão executadas as instruções do segundo bloco.

```
int numero = 3;
if (numero % 2 == 0)
{
    System.Console.WriteLine("O número é par.");
}
else
{
    System.Console.WriteLine("O número é ímpar.");
}
```

9.1.1 EXERCÍCIO DE SALA DE AULA

Crie um novo Projeto do tipo “Console Aplicattion” e nomei-o como “EstruturaDeControle1”.
Renomeie o arquivo “Program.cs” para “TestandoDoisValores.cs”.
O objetivo do nosso programa é receber dois valores e indicar qual deles é o maior.
Para tanto, insira o código marcado em negrito abaixo dentro do método principal “Main()”.

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

namespace EstruturaDeControle1
{
    class TestandoDoisValores
    {
      static void Main(string[] args)
      {
                // Testando o maior valor entre dois valores
                int num1, num2;

                Console.WriteLine(“Digite o primeiro valor: “);
                num1 = int.Parse(Console.ReadLine());
                Console.WriteLine(“Digite o segundo valor: “);
                num2 = int.Parse(Console.ReadLine());

                if (num1 > num2)
                {
                        Console.WriteLine(“O maior número é “ + num1);
                }
                else
                {
                        Console.WriteLine(“O maior número é “ + num2);
                }

                Console.ReadKey();
      }
    }
}
```

9.1.2 EXERCÍCIOS DE FIXAÇÃO

• Crie um programa que leia nome e idade de três pessoas, informando qual delas é a mais velha. Trabalhe com valores inteiros.

• Solicite que o usuário informe um valor e verifique se o mesmo é par ou ímpar.

• Solicite que o usuário informe um valor inteiro e verifique se o mesmo é negativo, zero ou positivo.

• Solicite que o usuário informe o nome e três notas de um aluno (com valores entre 1.0 a 10.0), calcule a sua média e teste as seguintes condições: se a média for menor ou igual a 5.0 exiba a mensagem “Reprovado”; se for menor ou igual a 7.0 exiba a mensagem “Atingiu a média”; se for menor ou igual a 8.9 exiba “Muito Bom”; se maior ou igual a 9.0, mostrar “Excelente”.

9.2 INSTRUÇÃO SWITCH

A instrução switch permite a execução condicional de instruções de acordo com o valor de um argumento teste, o qual pode ser uma variável, uma expressão numérica, uma string ou funções.

```
switch (<argumento-de-teste>)
{
    case <expressão-1>:
      // Código para quando argumento-de-teste = expressão-1.
      break;
    case <expressão-2>:
      // Código para quando argumento-de-teste = expressão-2.
      break;
    case <expressão-3>:
      // Código para quando argumento-de-teste = expressão-3.
      break;
    ...
    default:
      // Código para quando nenhum teste foi satisfeito.
      break;
}
```

9.2.1 EXERCÍCIO DE SALA DE AULA

Crie um novo Projeto do tipo “Console Aplicattion” com o nome de: “ExemploSwitch”.

Clique na classe padrão “Program.cs” e renomeie-a para “ExemploSwicth.cs”.

Logo em seguida, edite a mesma inserindo o código marcado em negrito abaixo dentro do método principal Main(),

O objetivo do programa é simular uma calculadora. Ele receberá dois valores e um operador, executará a operação e mostrará o resultado na tela.

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

namespace ExemploSwitch
{
    class ExemploSwitch
    {
      static void Main(string[] args)
      {
                double valor1, valor2, resultado;
                string operador;

                Console.WriteLine("Digite o primeiro valor: ");
                valor1 = double.Parse(Console.ReadLine());
                Console.WriteLine("Digite o segundo valor: ");
                valor2 = double.Parse(Console.ReadLine());
                Console.WriteLine("Escolha a operação(+, -, *, /): ");
                operador = Console.ReadLine();
```

```
    switch (operador)
                {
                        case "+":
                                resultado = valor1 + valor2;
                                Console.WriteLine("O resultado da soma é: "+
resultado);
                                break;
                        case "-":
                                resultado = valor2 - valor1;
                                Console.WriteLine("O resultado da subtração é: " +
resultado);
                                break;
                        case "*":
                                resultado = valor1 * valor2;
                                Console.WriteLine("O resultado da multiplicação é: " +
resultado);
                                break;
                        case "/":
                                resultado = valor1 / valor2;
                                Console.WriteLine("O resultado da divisão é: " +
resultado);
                                break;
                        default :
                                break;
                }

                Console.ReadKey();
          }
        }
    }
```

9.2.2 EXERCÍCIO DE FIXAÇÃO

• Usando a estrutura de condição Swicth, faça um programa que receba um valor e exiba sua descrição conforme abaixo:

- "C" para casado,
- "S" para solteiro,
- "D" para divorciado,
- "V" para viuvo

10 LOOPS

Os laços (loops) nos permitem executar tarefas de forma repetitiva dentro de um bloco de código. O C# possui três tipos de loops: loops contadores, loops codicionais e loops enumeradores.

10.1 INSTRUÇÃO FOR

Loops contadores executam uma tarefa um determinado número de vezes. A instrução for pode ser caracterizada como sendo loop contador, pois conhecemos os extremos que devem ser percorridos pelo bloco for.

```
for (<inicialização>; <teste>; <incremento>)
{
    // Código a ser executado enquanto o teste for
    // verdadeiro.
}
```

Parâmetro	Descrição
inicialização	Inicializa a variável de controle, usada para contar as execuções do loop.
teste	Teste que decide se o corpo do loop deve ou não ser executado.
incremento	Incrementa (ou decrementa) a variável de controle, preparando-a para a próxima execução do loop.

10.1.1 EXERCÍCIOS DE SALA DE AULA

Usando a estrutura de repetição for devemos criar um programa que imprima na tela os números de 1 a 10.

Crie um novo projeto do tipo "Console Aplicattion" com o nome de EstruturaDeForI, e em seguida renomeie a classe "Program.cs" para "ContarDezNumeros".

Então, dentro do método principal Main(), codifique conforme exemplo mostrado

abaixo.

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

namespace EstruturaDeForI
{
    class ContarDezNumeros
    {
      static void Main(string[] args)
      {
                for (int i = 1; i <= 10; i++)
                {
                        Console.WriteLine(i);
                }

                Console.ReadKey();
      }
    }
}
```

Usando a estrutura de repetição For vamos navegar e mostrar valores de um array de 5 posições.

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

namespace PercorrendoArray
{
    class PercorrerArray
    {
      static void Main(string[] args)
      {
                string[] nome = new string[5];

                nome[0] = “Rosa”;
                nome[1] = “Marcus”;
                nome[2] = “Ricardo”;
                nome[3] = “Pablo”;
                nome[4] = “Maria”;

                for (int i = 0; i < nome.Length; i++)
                {
                        Console.WriteLine(nome[i]);
                }

                Console.ReadKey();
      }
    }
}
```

10.1.2 EXERCÍCIOS DE FIXAÇÃO

• Crie um Array[10], peça para o usuário digitar valores do tipo string para preencher as 10 posições e em seguida percorra o mesmo mostrando todos os valores digitados.

• Crie um Array[5], peça para o usuário digitar valores inteiros para preencher as 5 posições e em seguida mostre somente o valor da posição [3].

Instrução while

A instrução while executa os comandos dentro do loop enquanto a condição que lhe serve como parâmetro for verdadeira. Sintaxe de instrução while:

```
while (<condição>)
{
    // Código a ser executado enquanto a condição for
    // verdadeira.
}
```

Se a condição for verdadeira as instruções colocadas dentro do bloco {} serão executadas, e o programa volta para o início do loop e novamente avaliará a condição. Logo que a condição se torna falsa, o programa passa a executar as instruções colocadas após as chaves (}).

Um loop while ou do pode ser encerrado pelas palavras-chave break, goto, return ou throw. A palavra-chave continue também pode ser usada.

10.1.3 EXERCÍCIOS DE SALA DE AULA

Para demonstrar o funcionamento da estrutura de laço while, o exemplo abaixo mostra como devemos fazer para imprimir na tela dez valores, iniciando pelo 1 até o 10.

Crie um novo projeto do tipo “Console Aplicattion” com o nome de “Estruturawhile”, e em seguida renomeie a classe “Program.cs” para “MostrarValores.cs”. Por fim, insira o código marcado em negrito abaixo no método Main().

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

namespace EstruturaWhile
{
    class MostrarValores
    {
      static void Main(string[] args)
      {
                int i = 1;
                while (i <= 10)
                {
                        Console.WriteLine(i);
                        i++;
                }

                Console.ReadKey();
      }
    }
}
```

Usando uma estrutura de laço while vamos criar um programa que leia a entrada somente de 5 valores.

Crie um novo projeto do tipo “Console Aplicattion”, com o nome de “EstruturaDeLacoI”.

Renomear a classe Principal “Program.cs” para “MostrandoCincoValores”.

Declare as possiveis variáveis, e estruture o código dentro do método principal Main().

Siga o exemplo abaixo:

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

namespace EstruturaDeLacoI
{
    class MostrandoCincoValores
    {
      static void Main(string[] args)
      {
                int num;
                int contador = 1;

                while (contador <= 5)
                {
                        Console.WriteLine(“Digite um número: “);
                        num = int.Parse(Console.ReadLine());

                        contador++;
                }

                Console.ReadKey();
      }
    }
}
```

10.1.4 EXERCÍCIOS DE FIXAÇÃO

- Escreva um programa que leia 10 valores fornecidos pelo usuário e mostre o número digitado e seu dobro.
- Gerar números de 0 a 50, variando de 5 em 5.Mostrar no final a soma de todos os números digitados.
- Escreva um programa que leia a entrada de 20 valore e mostre a soma dos valores positivos e a dos negativos.
- Escreva um programa que, a partir da entrada de um valor inteiro entre 1 e 10, exiba a tabuada desse número, do 0 ao 10.
- Ler 20 valores dois a dois. Testar estes valores. Se o segundo for maior que o primeiro, calcular a soma deles.
- Se o primeiro for maior que o segundo, calcular a diferença e, se forem iguais, multiplicá-los.
- Em uma empresa foram selecionados 12 funcionários novos, com diferentes idades e para cada uma delas foram registrados:
- nome, idade e mês de nascimento. Após o registro de dados, desejou-se saber:
- a idade da pessoa mais jovem;
- o nome e a idade da pessoa mais idosa;
- o número de pessoas nascida no mês de outubro(10).

10.2 INSTRUÇÃO DO

Você deve ter notado que na palavra-chave while a condição é avaliada antes de executar qualquer comando do loop. Se

a condição for verdadeira, as instruções do loop são executadas. Se a condição for falsa antes da primeira execução do loop, o programa prosseguirá com as instruções colocadas após o loop. Dessa forma, pode ser que as instruções no interior do loop jamais sejam executadas. Se você quiser que as instruções no interior do loop sejam executadas no mínimo uma vez, utilize a palavra-chave do. Isso garante que a instrução seja executada, pelo menos, uma vez, antes de avaliar se há necessidade de repetir ou não o loop. Sintaxe de instrução do:

```
do
{
    // Código a ser executado ao menos uma vez ou enquanto a
    // condição for verdadeira.
}
while (<condição>);
```

10.2.1 EXERCÍCIO DE SALA DE AULA

Para demonstrar o funcionamento da estrutura de laço do, o exemplo abaixo mostra que, ao contrário do que ocorre com a estrutura while, o bloco de código dentro do laço é executado ao menos uma vez, mesmo que a condição avaliada não seja válida.

Crie um novo projeto do tipo “Console Aplicattion” com o nome de “EstruturaDo”, e em seguida renomeie a classe “Program.cs” para “TestandoDo.cs”. Por fim, insira o código marcado em negrito abaixo no método Main().

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

namespace EstruturaDo
{
    class TestandoDo
    {
      static void Main(string[] args)
      {
                int i = 11;
                do
                {
                        Console.WriteLine(i);
                } while (i <= 10);

                Console.ReadKey();
      }
    }
}
```

10.3 INSTRUÇÃO FOREACH

Os loops de enumeração permitem percorrer itens de arrays e coleções. A sintaxe da instrução foreach é:

```
foreach (<tipo> <elemento> in <coleção>)
{
    // Código a ser executado uma vez para cada elemento da
    // coleção.
}
```

Parâmetro	Descrição
tipo	É o tipo de dados utilizado pela variável (elemento).
elemento	É a variável utilizada para percorrer os itens da coleção ou array.
coleção	É o objeto que contém o array ou coleção a ser iterado.

Exemplo:

```
string[] cores = new string[3];
cores[0] = “Azul”;
cores[1] = “Vermelho”;
cores[2] = “Verde”;

foreach (string cor in cores)
{
    System.Console.WriteLine(cor);
}
```

10.3.1 EXERCÍCIO DE SALA DE AULA

Usando a estrutura foreach vamos observar como percorrer uma coleção de dados, de uma maneira mais simples e rápida.

Crie um novo projeto do tipo “Console Aplicattion” com o nome “ExemploForEach” e em seguida renomeie a classe “Program.cs” para “ExemploForEach.cs”. Por fim, insira o código marcado em negrito abaixo dentro do método principal Main():

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

namespace ExemploForEach
{
    class Program
    {
      static void Main(string[] args)
      {
                string[] cores = new string[5];

                for (int i = 0; i < cores.Length; i++)
                {
                        Console.WriteLine("Digite o nome de uma cor: ");
                        cores[i] = Console.ReadLine();
                }

                foreach (string cor in cores)
                {
                        Console.WriteLine(cor);
                }

                Console.ReadKey();
      }
    }
}
```

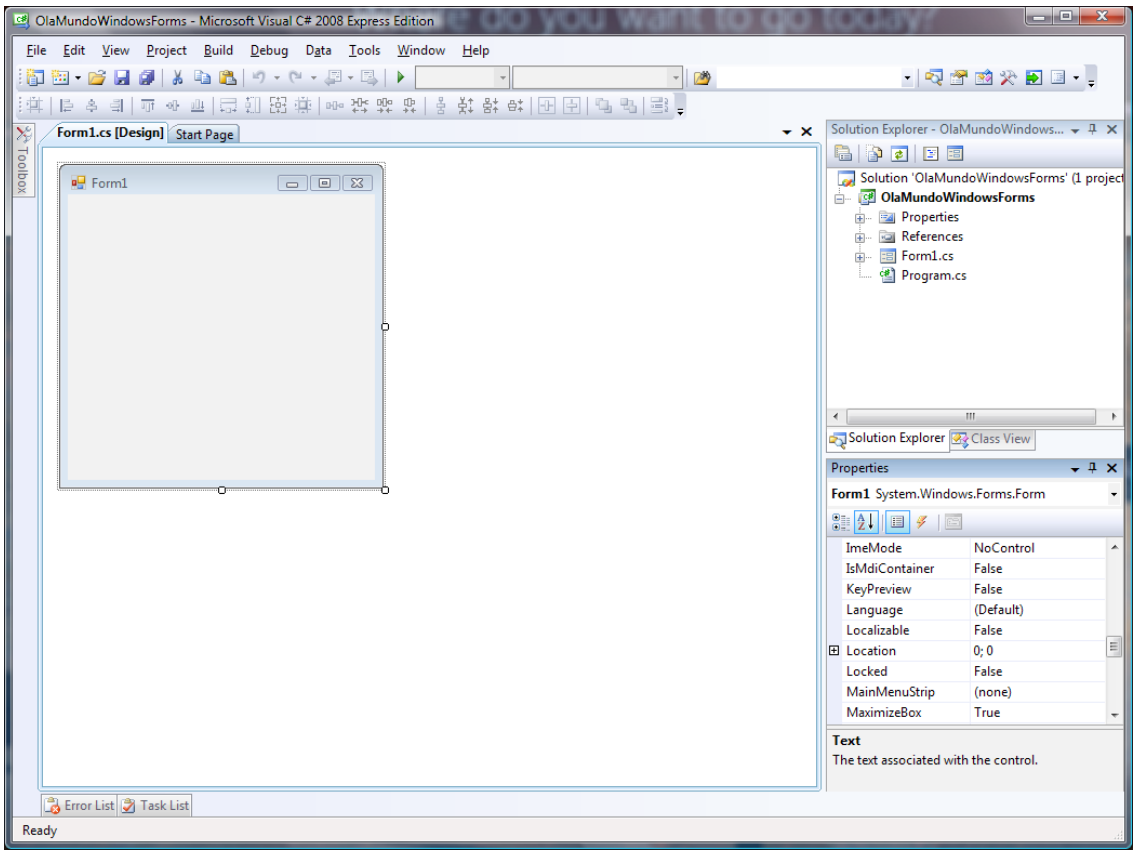
11 DESENVOLVENDO APLICAÇÕES GRÁFICAS

Se você achou fácil desenvolver aplicações no modo console em C#, vai se surpreender com a facilidade e versatilidade que o ambiente oferece para o desenvolvimento de aplicações gráficas (ou Windows Forms Applications, como elas são conhecidas no .NET Framework).

Sua Primeira Aplicação Windows Forms

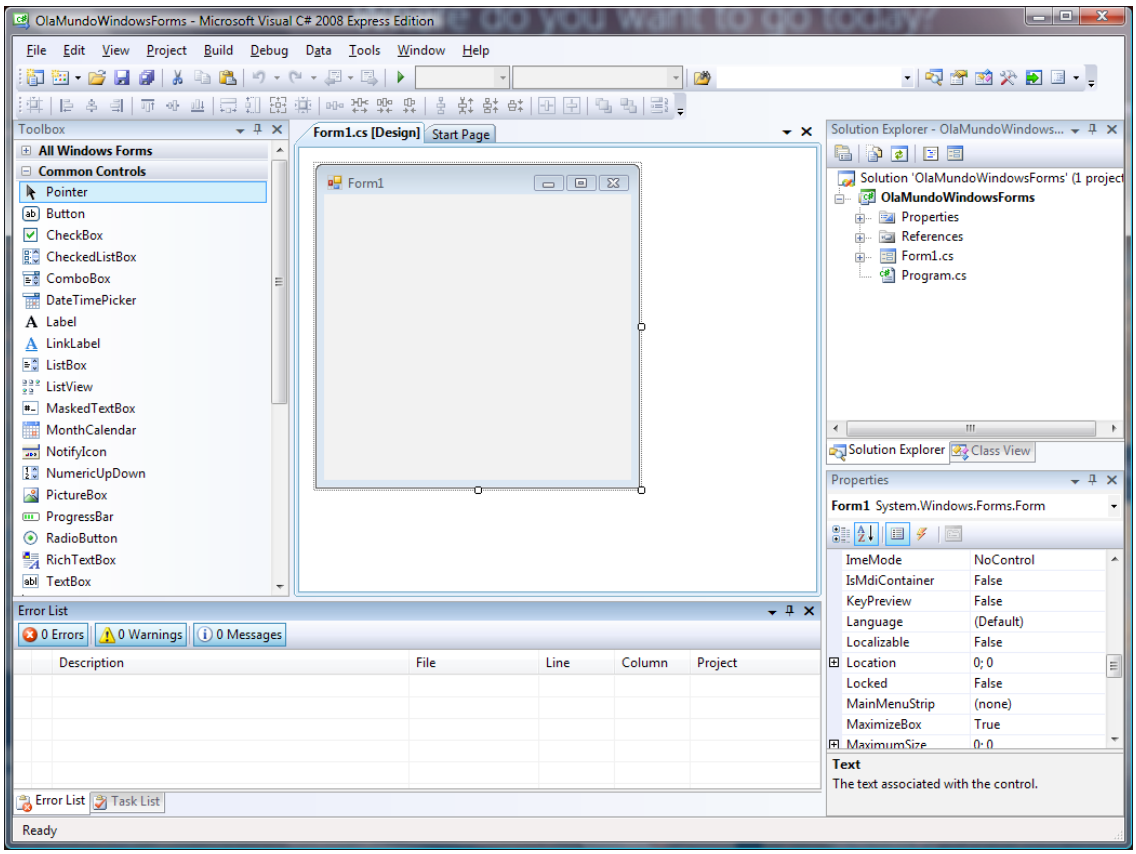
Para criar uma aplicação Windows Forms, primeiro clique em File > New Application... e selecione Windows Forms Application na caixa de templates, em seguida informando OlaMundoWindowsForms como nome da aplicação.

Ao clicar em OK você verá a seguinte tela preparada para que você projete seu formulário:



Criando uma Aplicação Windows Forms

Note que nos cantos da janela alguns botões reagem quando se coloca o mouse sobre eles, como por exemplo o botão "Toolbox" no canto esquerdo da janela. Para facilitar o processo de criação da tela, coloque o cursor do mouse sobre ele e, quando o painel do Toolbox se abrir, clique no botão "Auto Hide" para fixar a Toolbox na esquerda da janela. Repita o mesmo para os painéis Error List, Solution Explorer e Properties se necessário, e você obterá uma janela semelhante à mostrada abaixo



Janela de Edição do Projeto com os Principais Painéis Fixados

No centro da tela, como mencionado anteriormente, você vê o editor visual. É ali que desenharemos a nossa interface. Mas vale a pena darmos uma explorada nos diferentes painéis, para que nos familiarizemos.

Na esquerda, a Toolbox (caixa de ferramentas) é a caixa que nos oferece os controles que poderemos colocar na interface que estamos criando. Abaixo dela temos dois painéis intercambiáveis, Error List e Task List. Em Error List serão exibidos os erros de compilação ocorridos ao gerar o nosso projeto. Já em Task List podemos criar pequenos lembretes para nós mesmos, como anotações sobre pontos pendentes no projeto e assim por diante.

À direita, acima, temos outros dois painéis intercambiáveis, Solution Explorer e Class View. Os dois nos oferecem diferentes visões dos arquivos que criamos para o nosso projeto, e uma rápida olhadela já nos permite ver que o C# cria muita coisa automaticamente para nós assim que um novo projeto é criado.

E logo abaixo dos painéis Solution Explorer e Class View existe um outro painel muito importante, o Properties, que exibe as propriedades do objeto selecionado no Designer ou nos painéis Solution Explorer e Class View. É por meio dele que definiremos os atributos dos controles que vamos adicionar à nossa interface.

11.2 CONTROLES

Na seção anterior mencionamos a palavra “controles” várias vezes, mas ainda não explicamos o seu significado. No Visual Studio, um controle é um componente de interface, como por exemplo um botão ou uma caixa de texto. O Visual Studio nos oferece um editor visual que nos permite facilmente criar telas por meio do arrastar e soltar de controles, e depois pela sua configuração através do painel de propriedades. Vamos brincar com isso em seguida.

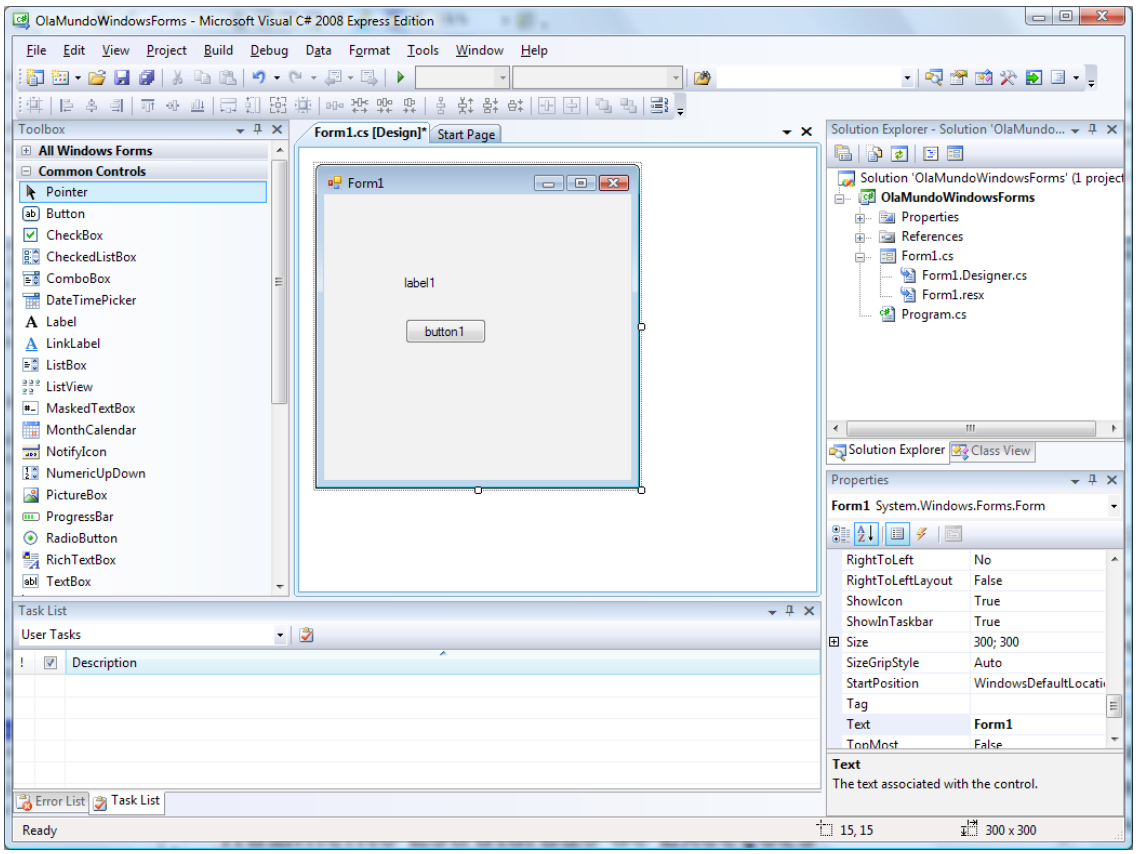
11.3 DESENHANDO UMA JANELA

Como você pode ver, assim que escolhe criar uma aplicação Windows Forms o Visual Studio cria uma janela inicial para você. Isso porque todas as aplicações Windows Forms terão pelo menos uma janela. Como veremos em seguida, uma janela é um controle também, então tudo o que dissemos na seção anterior se aplica para janelas também. Elas também têm propriedades e eventos.

Para experimentar, vamos criar a versão gráfica da nossa aplicação OlaMundo.

Para isso usaremos a Toolbox. Note que ela vem dividida em várias seções, como Common Controls, Containers e assim por diante. Estas seções separam os controles em categorias, então fica fácil de vermos como os diferentes controles se relacionam. Por ora expanda a seção Common Controls, pois ela agrupa os controles que usaremos com mais freqüência.

Fazendo uso da Toolbox, adicione dois controles à janela vazia no centro do editor. A maneira de fazer isso é você que escolhe: você pode clicar no controle que deseja adicionar no Toolbox e depois clicar no ponto da janela em que deseja que o controle seja adicionado; ou você pode arrastar o controle da Toolbox até a janela. Experimente brincar com o editor, você verá que ele lhe oferece muitas facilidades para adicionar controles e alinhá-los consistentemente. Depois de brincar, tente deixar a janela semelhante à mostrada abaixo:



Criando uma Janela

Pronto, a parte difícil está pronta (-:

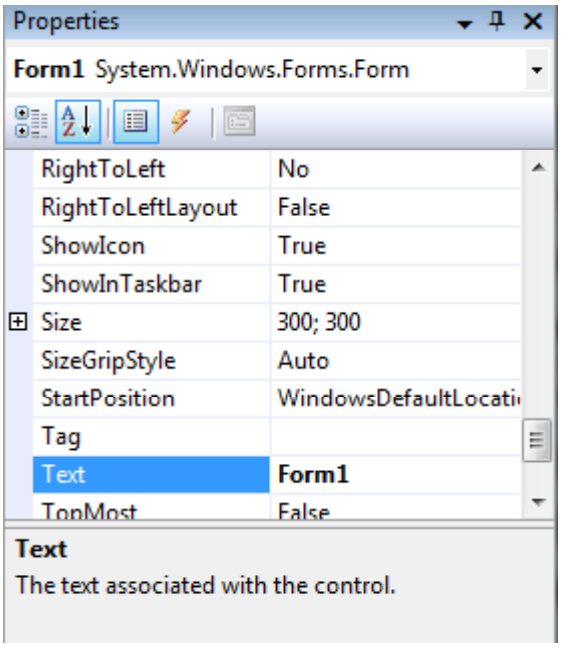
Agora temos que ajustar os controles que adicionamos às nossas necessidades. Por exemplo, vamos querer mudar o título da janela e os textos mostrados pelo label e botão. É ai que entra o trabalho no painel de Propriedades.

11.4 ALTERANDO AS PROPRIEDADES DOS CONTROLES

Com o básico da tela desenhado, chegou a hora de fazer os ajustes finos. Nesta etapa faremos os diferentes controles mostrarem as informações que desejamos, assim como podemos fazer outros ajustes na parte visual, como cores e outras firulas.

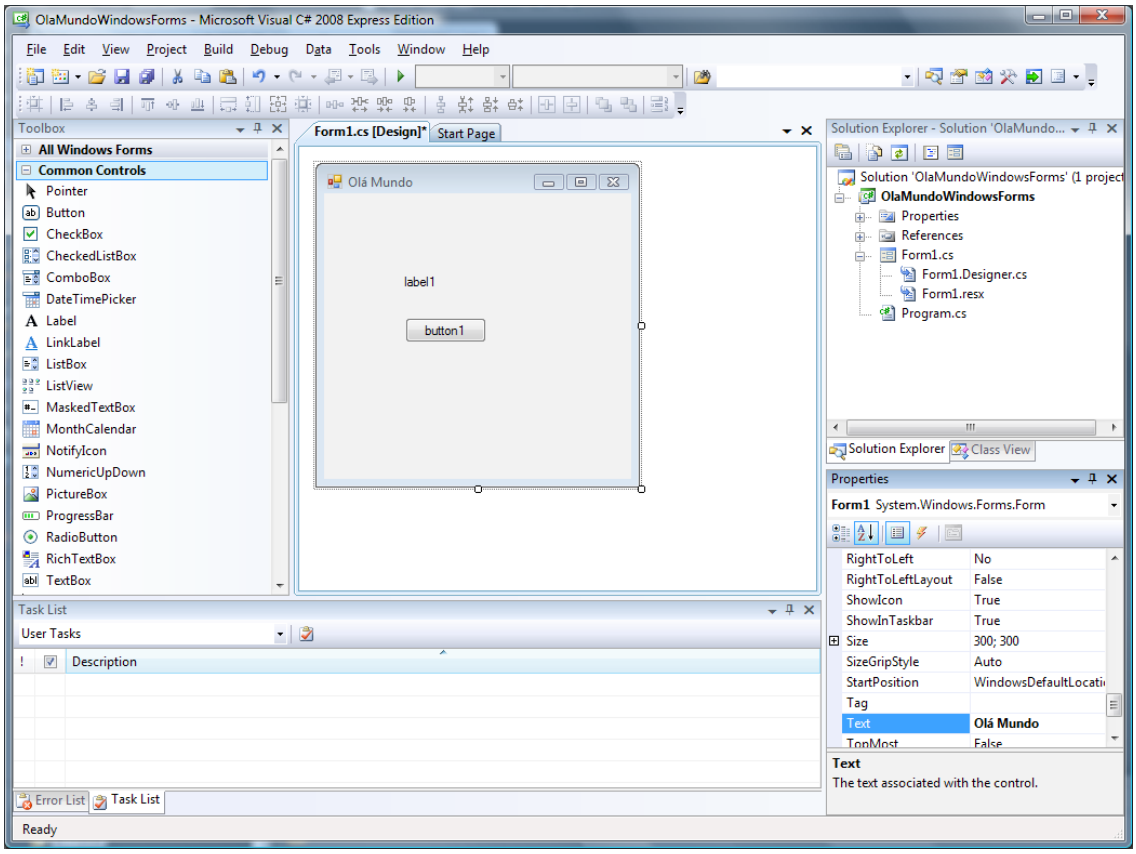
No nosso exemplo, bem simples, vamos nos concentrar em alterar os textos exibidos pelos três controles que temos no nosso formulário: a própria janela, o label e o botão.

Para alterar as propriedades de um objeto, selecione-o no editor e atente no canto inferior direito da tela, no painel Propriedades. Comece clicando na janela, por exemplo, e repare nas propriedades listadas para ela:



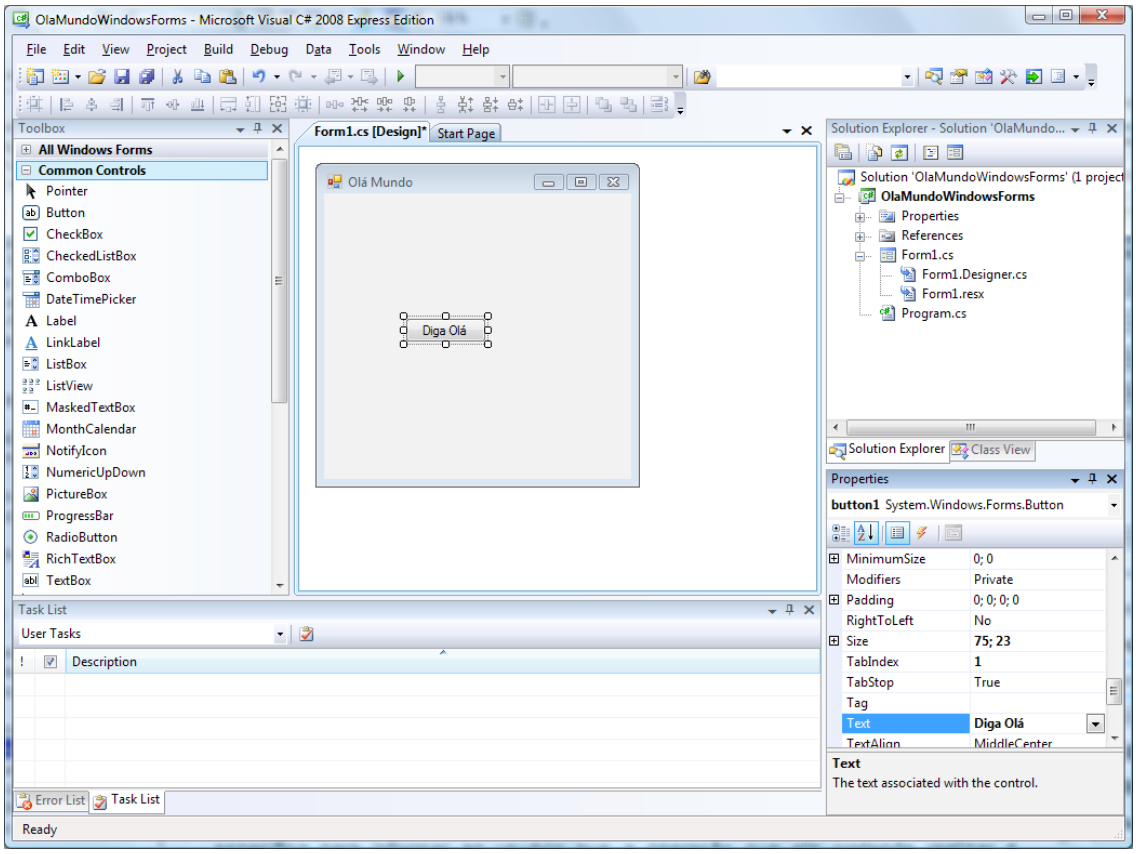
Propriedades de um Form

A principal propriedade de um Form é a Text. É ela que indica o texto a ser mostrado na barra de títulos da janela. Para alterar o título da nossa janela basta editar a propriedade. Troque-a por "Ola Mundo!", por exemplo. Note que ao pressionar Enter o título da janela muda de acordo com o valor informado:



11.5 ALTERANDO O TÍTULO DE UMA JANELA

Repita o processo para os outros dois componentes, mudando a propriedade Text do label para um texto vazio e a mesma propriedade do botão para “Diga Olá”, conforme mostrado na tela abaixo:



Nosso Formulário editado

OK, a parte visual da nossa primeira aplicação está pronta. Agora precisamos adicionar-lhe um pouco de inteligência ...

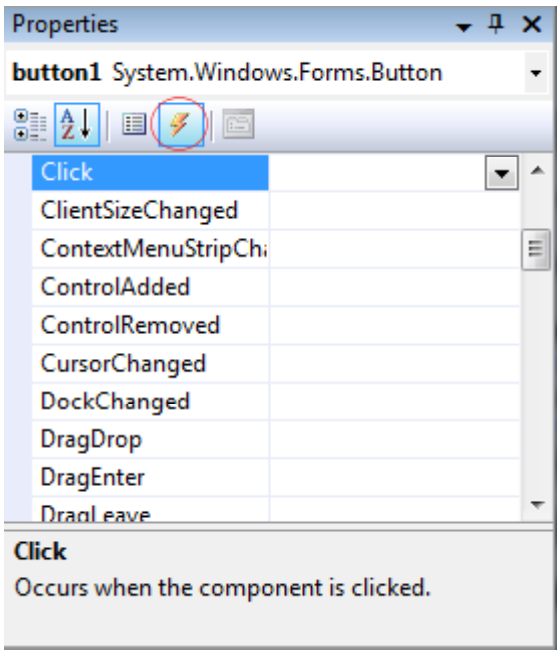
11.6 TRATANDO EVENTOS

O C#, assim como outras linguagens usadas para desenvolver aplicativos baseados em janelas, utiliza um conceito chamado de Orientação por Eventos. Em essência isso quer dizer que programamos nosso aplicativo como uma série de respostas aos eventos gerados pelo usuário (ou mesmo pelo próprio aplicativo). Estes eventos são ações tomadas pelo usuário, por exemplo, como clicar em um botão ou entrar um texto em uma caixa de textos.

Em C#, toda vez que o usuário efetua uma ação, um código de tratamento de eventos é automaticamente lançado. Tudo o que precisamos fazer é dizer ao programa o que queremos que ele faça quando cada um destes eventos ocorrer, e a plataforma toma conta do resto por nós.

Diferentes controles lidam com diferentes eventos. Um botão precisa responder quando é clicado, um formulário precisa responder quando é maximizado, uma caixa de textos precisa responder quando o texto nela contido é alterado e assim por diante. Então, para definirmos o tipo de resposta a cada um destes eventos, usaremos uma conjunção do painel de Propriedades com o Editor de Código.

Funciona mais ou menos assim: no caso do aplicativo que estamos criando, queremos que, quando o usuário clicar no botão “Diga Olá”, uma mensagem “Olá mundo!” seja exibida na tela. Assim, precisamos responder ao evento de clique no botão: quando ele ocorrer, mostraremos um texto no label.

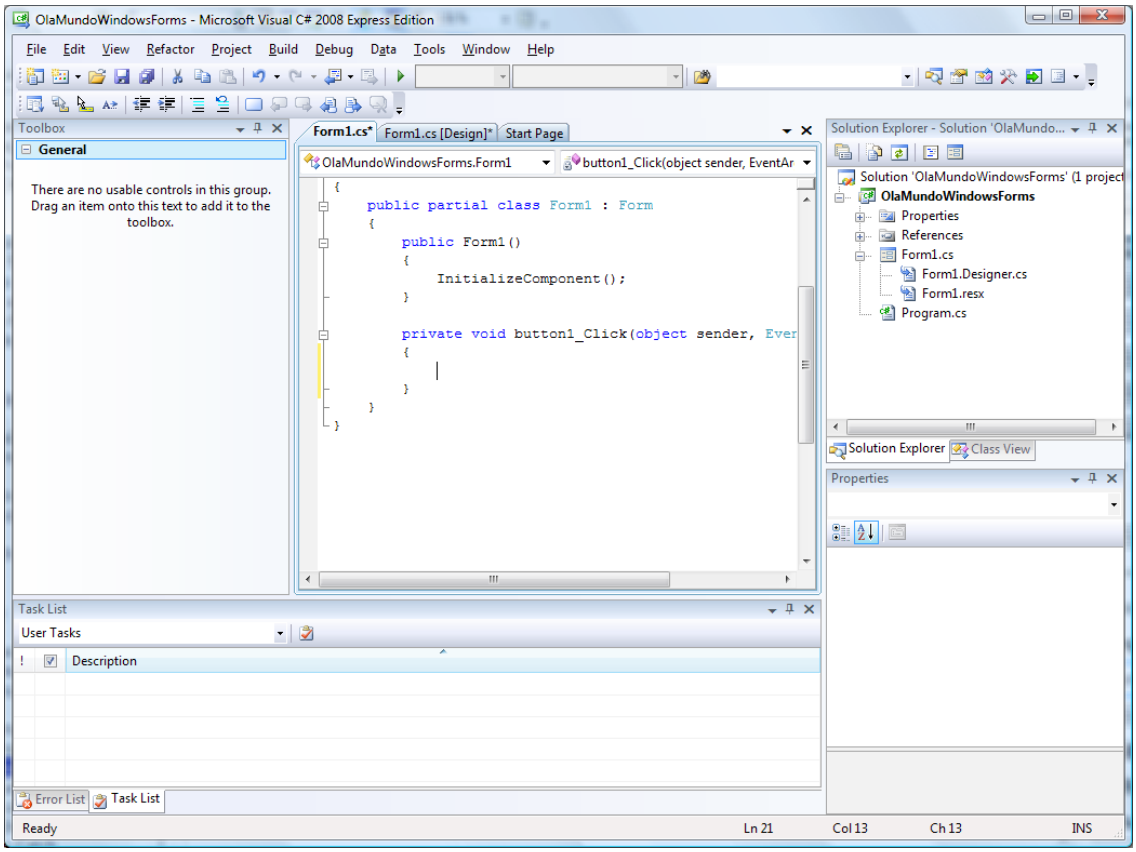


O Painel Propriedades exibindo os Eventos do botão

Para ver os eventos possíveis em um botão, primeiro selecione o botão no editor e depois, no painel de propriedades, clique no botão Events. Para voltar a ver as propriedades do botão clique no botão Properties.

Ao clicar no botão Events para um botão você verá a lista com todos os eventos passíveis de serem capturados por um botão. Obviamente o mais comum deles é o evento Click, que ocorre quando o usuário clica no botão.

Para editar o código que tratará o evento Click do nosso botão, clique duas vezes sobre o evento no painel de propriedades. O Visual Studio o levará para o Editor de Código, onde você poderá descrever o que quer que aconteça quando o botão for clicado.



Editando o evento Click do botão

Na figura acima você pode ver que o Visual Studio já cria muita coisa para você, faltando apenas escrever o que fazer efetivamente para tratar o evento.

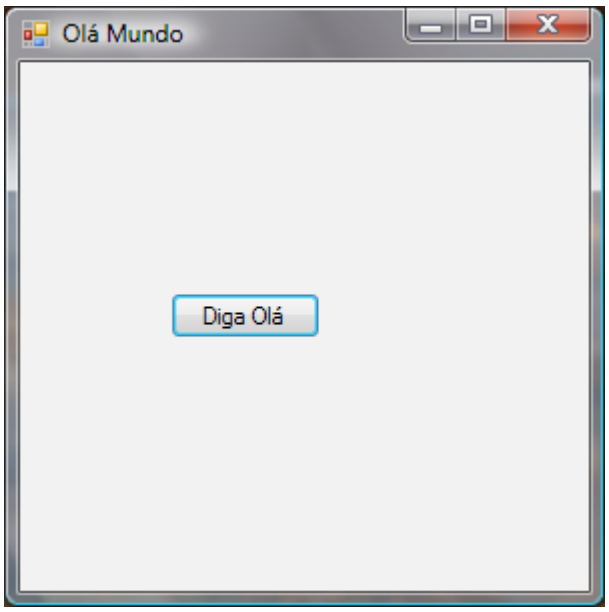
Altere o código no editor acrescentando a linha em negrito mostrada abaixo:

```
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;

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

      private void button1_Click(object sender, EventArgs e)
      {
            label1.text = “Olá Mundo!”;
      }
    }
}
```

Pronto, sua primeira aplicação visual está pronta, e você só precisou usar seus dotes artísticos e escrever uma única linha de código ...

Clique em File > Save All... para salvar o seu projeto, e para executá-lo clique em Debug > Start Without Debugging.

O resultado deve ser semelhante ao mostrado abaixo:



Olá Mundo em Windows Forms

Agora clique no botão e surpreenda-se com o seu sucesso ...

11.6 MÉTODOS

Em determinadas situações poderemos querer realizar certas operações com os controles de uma janela, como por exemplo ocultar um botão ou selecionar um checkbox. Estas operações estão disponíveis por meio de métodos associados aos controles. Podemos entender um método como uma mensagem que enviaremos a um controle, solicitando que ele faça algo.

No nosso exemplo anterior, poderíamos modificar o código para ocultar o botão Diga Olá uma vez que ele foi clicado. É fácil de fazer, como você verá. Edite o código do programa adicionando a linha em negrito.

```
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;

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

      private void button1_Click(object sender, EventArgs e)
      {
            label1.text = "Olá Mundo!";
            button1.hide();
      }
    }
}
```

No exemplo acima estamos chamando o método hide() do controle button1. A notação é sempre esta: o objeto que se quer acessar, um ponto e o nome do método a ser chamado, seguido de parênteses. Métodos podem receber parâmetros.

Parece familiar? Exatamente, métodos não passam de funções. A única diferença é que os métodos estão associados aos objetos.

11.7 NOMEANDO OS CONTROLES

Se você seguiu os exemplos anteriores, percebeu como pode às vezes pode ser difícil lidar com muitos controles na tela. Trabalhar com label1 e button1 pode não ter sido muito complicado num exemplo simples como o nosso, mas imagine quando o número de controles se multiplica em um formulário apenas mediano. Aí estes nomes "artificiais" não são nenhum pouco intuitivos. O melhor nestas situações é nomear você mesmo os componentes, pelo menos aqueles aos quais você precisará se referenciar em meio ao código.

Por exemplo, compare este trechos de código do exemplo anterior ...

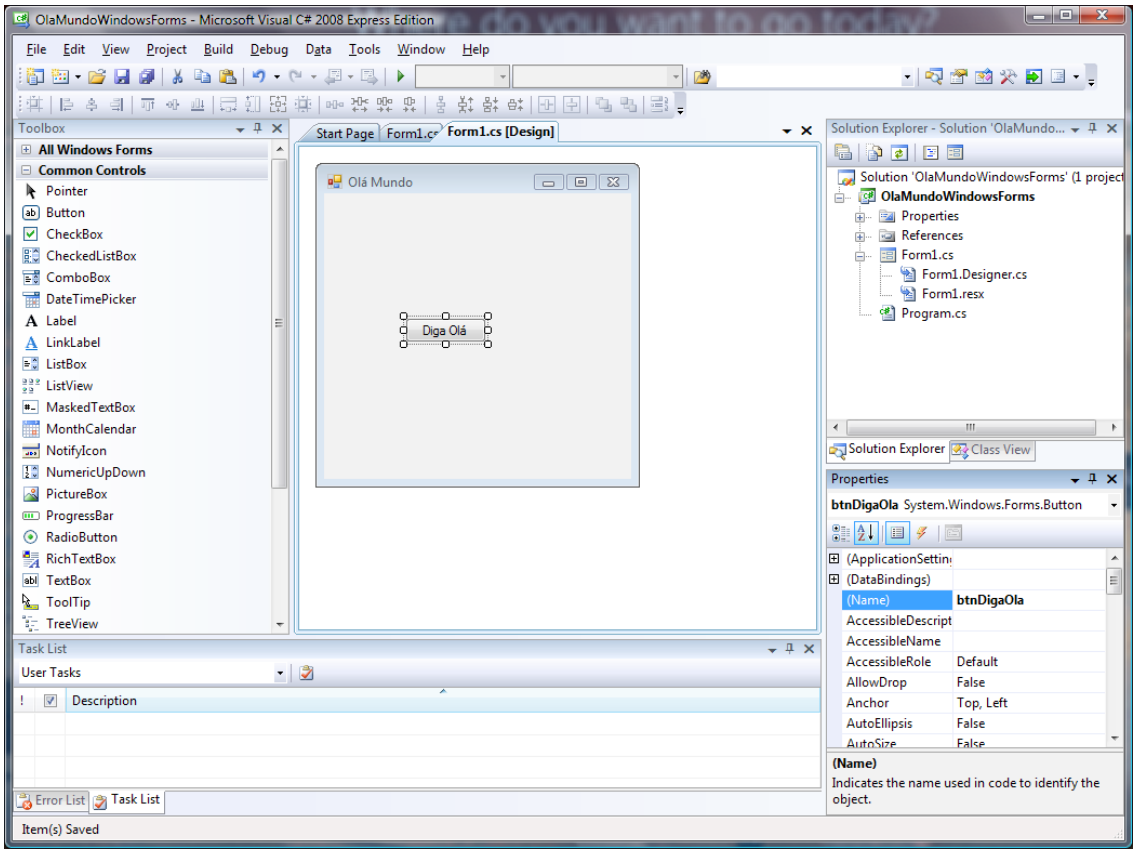
```
private void button1_Click(object sender, EventArgs e)
{
    label1.text = "Olá Mundo!";
    button1.hide();
}
```

com este:

```
private void btnDigaOla_Click(object sender, EventArgs e)
{
    lblTextoOla.text = "Olá Mundo!";
    btnDigaOla.hide();
}
```

A maneira de conseguir isto é através do próprio painel de propriedades. Lá, bem no topo da lista de propriedades de cada controle, você encontrará uma propriedade chamada **(Name)**. Basta mudá-la para que o nome do componente mude, mesmo em trechos de código que você já tenha escrito, o que facilita bastante...

É importante notar que a propriedade Name segue as mesmas regras de nomes de variáveis. Na verdade, podemos até mesmo entender um botão colocado em um formulário como uma variável do tipo botão.



Alterando o nome de um controle.

12 USANDO OS CONTROLES

Como pudemos ver no início deste capítulo, a Toolbox oferece uma grande variedade de controles, que ficam à sua disposição para serem usados em suas aplicações. Não temos tempo nem espaço para vermos todos eles, então vamos explorar apenas alguns dos principais. Fique à vontade para experimentar os outros que julgar interessantes.

12.1 FORM

O controle Form, ou formulário, representa uma janela da nossa aplicação, e normalmente será preenchido com uma série de outros controles.

Principais Propriedades

Propriedade	Descrição
AcceptButton	Permite indicar o botão que será ativado ao se pressionar Enter quando o formulário tem o foco do teclado.
BackColor	Permite indicar a cor de fundo da janela.
BackgroundImage	Permite indicar uma imagem para ser usada como fundo da janela.
BackgroundImageLayout	Permite indicar a maneira como a imagem de fundo será usada para preencher a janela.
Bounds	Indica o tamanho e a localização do formulário em relação ao seu controle pai.
CancelButton	Permite indicar o botão que será ativado ao se pressionar Esc quando a janela tem o foco do teclado.
ContextMenuStrip	Indica o menu de contexto a exibir quando o usuário clica com o botão direito sobre o formulário.
ControlBox	Permite indicar se a janela terá ou não uma caixa de controle no canto esquerdo da sua barra de título.
Cursor	Permite indicar o cursor a usar quando o ponteiro do mouse é posicionado sobre a janela.
Enabled	Permite habilitar e desabilitar o formulário.
Focused	Indica se este formulário tem ou não o foco.
FormBorderStyle	Permite indicar o tipo de borda da janela.
Icon	Permite indicar o ícone associado a esta janela.
IsMDIContainer	Indica se este formulário é um contêiner MDI (multi-document interface).
Location	Indica as coordenadas do canto superior esquerdo da janela.
MainMenuStrip	Permite especificar a barra de menus a ser usada por esta janela.
MaximizeBox	Permite indicar se o botão de maximizar estará visível na barra de título da janela.
MinimizeBox	Permite indicar se o botão de minimizar estará visível na barra de título da janela.
Opacity	Indica a opacidade da janela.
ShowIcon	Indica se o ícone da janela deve estar visível na barra de título.
Size	Indica o tamanho do formulário.
Text	Indica o título da janela.
Visible	Indica se o formulário está visível ou não.
WindowState	Indica o estado da janela (normal, minimizada ou maximizada).

Principais Eventos

Evento	Descrição
Activated	Lançado quando o formulário é ativado via código ou diretamente pelo usuário.
Click	Lançado quando o usuário clica no formulário.
Closed	Lançado quando o formulário é fechado.
Closing	Lançado quando o formulário está sendo fechado.
Deactivated	Lançado quando o formulário perde o foco e não é mais o form ativo.
DoubleClick	Lançado quando o formulário recebe um duplo clique.
EnabledChanged	Lançado quando o formulário é habilitado ou desabilitado.
GotFocus	Lançado quando o formulário recebe o foco.
Load	Lançado quando o formulário é exibido pela primeira vez desde que a aplicação foi iniciada.
LostFocus	Lançado quando o formulário perde o foco.
Resize	Lançado quando o tamanho do formulário é modificado.
TextChanged	Lançado quando o título da janela é alterado.
VisibleChanged	Lançado quando a propriedade Visible é alterada.

Principais Métodos

Método	Descrição
Activate	Ativa o formulário e lhe dá foco.
BringToFront	Traz o formulário para frente.
Close	Fecha o formulário.
Dispose	Libera os recursos usados pelo formulário.
Focus	Dá o foco ao formulário.
Hide	Oculta o formulário.
SendToBack	Envia o formulário para trás.
Show	Exibe o formulário.
ShowDialog	Exibe o formulário como uma caixa de diálogos modal.

12.2 COMMON CONTROLS

O grupo Common Controls reúne aqueles que são os controles mais freqüentemente utilizados. Aqui encontraremos botões, caixas de texto e labels.

12.3 BUTTON

O controle Button é provavelmente um dos controles mais simples que você encontrará na Toolbox, mas ao mesmo tempo ele é um dos mais úteis. Ele nos oferece um botão simples que realiza uma determinada ação quando clicado pelo usuário.

Principais Propriedades

Propriedade	Descrição
BackgroundImage	Permite indicar a imagem de fundo do botão.
BackgroundImageLayout	Permite indicar o tipo de layout a usar quando exibindo a imagem de fundo do botão.
Cursor	Permite indicar o cursor que aparece quando o mouse é colocado sobre o botão.
Enabled	Indica se o botão está habilitado ou não.
Font	Indica a fonte usada para exibir o texto do botão.
Image	Indica a imagem dentro do botão.
ImageAlign	Indica o tipo de alinhamento da imagem dentro do botão.
Size	Indica as dimensões do botão (Width é a largura, Height é a altura).
TabStop	Indica se o usuário pode usar a tecla Tab para dar foco no botão.
Text	Indica o texto mostrado pelo botão.
TextAlign	Indica o alinhamento do texto dentro do botão.
Visible	Indica se o botão está visível ou não.

Principais Eventos

Evento	Descrição
Click	Lançado quando o botão é clicado.
EnabledChanged	Lançado quando o botão é habilitado ou desabilitado programaticamente (veja a propriedade enabled).
VisibleChanged	Lançado quando o botão é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	Descrição
BringIntoView	Tenta trazer o botão para a área visível da tela, rolando o seu contêiner se necessário/possível.
Focus	Tenta colocar o foco do teclado neste botão.
MoveFocus	Tenta retirar o foco do teclado deste botão, movendo-o para o próximo controle ou o controle anterior no contêiner.

CheckBox

O controle CheckBox oferece a possibilidade de o usuário ativar ou desativar uma determinada opção. Este é um componente bastante útil para aqueles casos em que queremos que o usuário ligue ou desligue uma dada situação ou informe se um elemento está ativado ou não.

Principais Propriedades

Propriedade	Descrição
BackColor	Determina a cor de fundo do checkbox.
Checked	Determina se o checkbox está marcado ou não.
CheckState	Determina o estado do checkbox.
Enabled	Determina se o controle está habilitado.
Focused	Determina se o controle tem o foco.
Font	Permite indicar a fonte a ser usada pelo checkbox.
TabIndex	Lê ou altera a ordem de foco deste controle dentro do seu controle-pai.
Text	O texto exibido por este checkbox.
Visible	Permite indicar se o checkbox deve estar visível ou não.

Primcipais Eventos

Evento	Descrição
Click	Lançado quando o checkbox é clicado.
EnabledChanged	Lançado quando o checkbox é habilitado ou desabilitado programaticamente (veja a propriedade enabled).
GotFocus	Lançado quando o checkbox recebe o foco.
LostFocus	Lançado quando o checkbox perde o foco.
VisibleChanged	Lançado quando o checkbox é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	Descrição
BringToFront	Traz este controle para frente.
Focus	Dá foco no checkbox.
Hide	Oculta o checkbox.
SendToBack	Envia este controle para trás.
Show	Exibe este controle.

12.4 LABEL

Um label oferece uma maneira conveniente de exibir uma informação ao usuário.

Principais Propriedades

Propriedade	Descrição
BackColor	Determina a cor de fundo do label.
Enabled	Determina se o controle está habilitado.
Font	Permite indicar a fonte a ser usada pelo checkbox.
Text	O texto exibido por este label.
TextAlign	Indica o alinhamento de texto neste controle.
Visible	Permite indicar se o checkbox deve estar visível ou não.

Primcipais Eventos

Evento	Descrição
VisibleChanged	Lançado quando o checkbox é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	Descrição
BringToFront	Traz este controle para frente.
Hide	Oculta o label.
SendToBack	Envia este controle para trás.
Show	Exibe este controle.

12.5 LISTBOX

O controle ListBox apresenta uma lista de opções e permite que o usuário selecione uma ou várias delas.

Principais Propriedades

Propriedade	Descrição
BackColor	Determina a cor de fundo do listbox.
Enabled	Determina se o controle está habilitado.
Focused	Determina se o controle tem o foco.
Font	Permite indicar a fonte a ser usada pelo checkbox.
Items	Dá acesso à lista de opções contidas neste ListBox.
SelectedIndex	Indica o índice da opção selecionada, se houver.
SelectedIndices	Indica os índices das opções selecionadas, se houver.
SelectedItem	Lê ou altera a opção selecionada.
SelectredItems	Lê a lista de opções selecionadas.
TabIndex	Lê ou altera a ordem de foco deste controle dentro do seu controle-pai.
Text	Indica o texto da opção correntemente selecionada, se houver uma única.
Visible	Permite indicar se o checkbox deve estar visível ou não.

Primcipais Eventos

Evento	Descrição
EnabledChanged	Lançado quando o listbox é habilitado ou desabilitado programaticamente (veja a propriedade enabled).
GotFocus	Lançado quando o listbox recebe o foco.
LostFocus	Lançado quando o listbox perde o foco.
SelectedIndexChanged	Lançado quando a propriedade SelectedIndex ou SelectedIndices muda.
SelectedValueChanged	Lançado quando a propriedade SelectedValue muda.
VisibleChanged	Lançado quando o checkbox é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	Descrição
BringToFront	Traz este controle para frente.
Focus	Dá foco no checkbox.
Hide	Oculta o checkbox.
SendToBack	Envia este controle para trás.
Show	Exibe este controle.

12.6 PICTUREBOX

O controle PictureBox oferece o recurso de exibir imagens dentro de um formulário

Principais Propriedades

Propriedade	Descrição
BackColor	Determina a cor de fundo do picturebox.
Enabled	Determina se o controle está habilitado.
Focused	Determina se o controle tem o foco.
Image	Permite ler ou indicar a imagem a ser exibida dentro do controle.
ImageLocation	Permite ler ou indicar o caminho ou URL até a imagem a ser exibida pelo controle.
TabIndex	Lê ou altera a ordem de foco deste controle dentro do seu controle-pai.
Visible	Permite indicar se o picturebox deve estar visível ou não.

Primcipais Eventos

Evento	Descrição
Click	Lançado quando o picturebox é clicado.
DoubleClick	Lançado quando o picturebox recebe um duplo clique.
EnabledChanged	Lançado quando o picturebox é habilitado ou desabilitado programaticamente (veja a propriedade enabled).
GotFocus	Lançado quando o picturebox recebe o foco.
LostFocus	Lançado quando o picturebox perde o foco.
VisibleChanged	Lançado quando o checkbox é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	Descrição
BringToFront	Traz este controle para frente.
Focus	Dá foco no picturebox.
Hide	Oculta o picturebox.
Load	Exibe uma imagem no picturebox.
SendToBack	Envia este controle para trás.
Show	Exibe este controle.

12.7 RADIOBUTTON

O controle RadioButton oferece uma série de opções das quais o usuário pode selecionar apenas uma.

Principais Propriedades

Propriedade	Descrição
BackColor	Determina a cor de fundo do checkbox.
Checked	Determina se o radiobutton está marcado ou não.
Enabled	Determina se o controle está habilitado.
Focused	Determina se o controle tem o foco.
Font	Permite indicar a fonte a ser usada pelo checkbox.
TabIndex	Lê ou altera a ordem de foco deste controle dentro do seu controle-pai.
Text	O texto exibido por este radiobutton.
Visible	Permite indicar se o checkbox deve estar visível ou não.

Primcipais Eventos

Evento	Descrição
CheckedChanged	Lançado quando o atributo Checked muda.
Click	Lançado quando o radiobutton é clicado.
EnabledChanged	Lançado quando o radiobutton é habilitado ou desabilitado programaticamente (veja a propriedade enabled).
GotFocus	Lançado quando o radiobutton recebe o foco.
LostFocus	Lançado quando o radiobutton perde o foco.
VisibleChanged	Lançado quando o radiobutton é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	Descrição
BringToFront	Traz este controle para frente.
Focus	Dá foco no radiobutton.
Hide	Oculta o radiobutton.
SendToBack	Envia este controle para trás.
Show	Exibe este controle.

12.8 TEXTBOX

O controle TextBox oferece a maneira mais básica de entrada de dados para aplicações baseadas em Windows Forms.

Principais Propriedades

Propriedade	Descrição
AcceptReturn	Determina se este textbox pode receber caracteres de quebra de linha.
AcceptTab	Determina se este textbox pode receber caracteres de tabulação.
BackColor	Determina a cor de fundo do textbox.
CanUndo	Determina se o usuário pode desfazer a última ação dentro deste controle.
Enabled	Determina se o controle está habilitado.
Focused	Determina se o controle tem o foco.
Font	Permite indicar a fonte a ser usada pelo textbox.
Modified	Determina se o texto armazenado por este controle foi modificado pelo usuário.
Multilline	Determina se este textbox pode receber múltiplas linhas de texto.
PasswordChar	Lê ou altera o caractere usado para ocultar senhas em um textbox de uma única linha.
ReadOnly	Indica se este textbox é de apenas leitura.
ScrollBars	Indica se este componente deve mostrar barras de rolagem.
SelectedText	Lê ou indica o texto a aparecer selecionado dentro do componente,

SelectionLength	Indica o número de caracteres selecionados dentro deste controle.
SelectionStart	Indica o índice do primeiro caractere selecionando dentro deste controle.
TabIndex	Lê ou altera a ordem de foco deste controle dentro do seu controle-pai.
Text	O texto exibido por este textbox.
TextAling	Indica o tipo de alinhamento de texto dentro deste controle.
TextLength	Indica o número de caracteres armazenados dentro deste controle.
Visible	Permite indicar se o textbox deve estar visível ou não.
WordWrap	Indica se o componente deve quebrar as linhas automaticamente sempre que necessário.

Primcipais Eventos

Evento	**Descrição**
EnabledChanged	Lançado quando o textbox é habilitado ou desabilitado programaticamente (veja a propriedade enabled).
GotFocus	Lançado quando o textbox recebe o foco.
KeyPress	Lançado toda a vez que uma tecla é pressionada dentro do controle.
LostFocus	Lançado quando o textbox perde o foco.
TextChanged	Lançado quando o texto armazenado pelo textbox é modificado.
VisibleChanged	Lançado quando o textbox é exibido ou ocultado programaticamente (veja o método hide() e a propriedade Visible).

Principais Métodos

Método	**Descrição**
BringToFront	Traz este controle para frente.
Cut	Move o texto atualmente selecionado no controle para a área de transferência.
Focus	Dá foco no textbox.
Hide	Oculta o textbox.
Paste	Substitui o texto atualmente selecionado no controle por aquele armazenado na área de transferência.
Select	Ativa este textbox.
SelectAll	Seleciona todo o texto dentro deste textbox.
SendToBack	Envia este controle para trás.
Show	Exibe este controle.

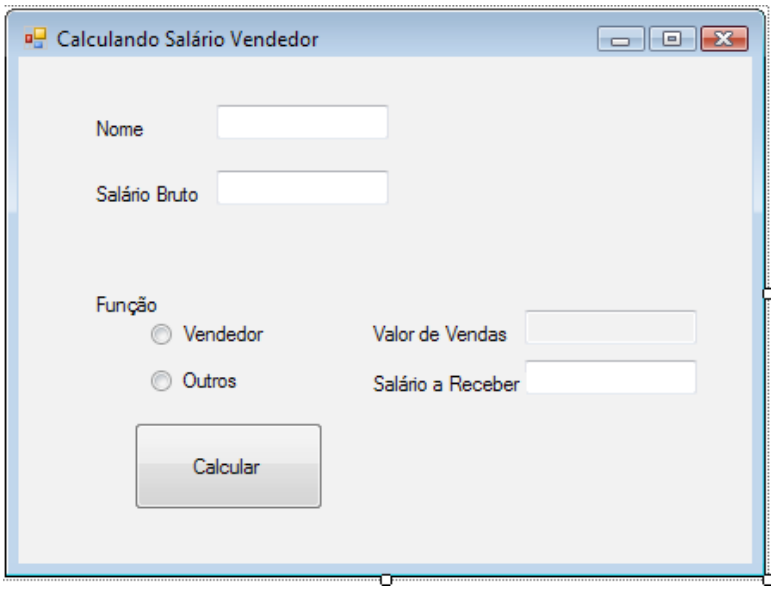
12.9 CÁLCULO DE SALÁRIO

O programa irá calcular o salário de um funcionário a partir da entrada do valor de salário bruto, sua função e, se pertinente, o valor das duas vendas.

Se o funcionário for Vendedor uma comissão de 10% sobre o total de suas vendas deverá ser adicionada ao seu salário.

Passos a serem seguidos:

- Crie um novo Projeto Windows Forms: e nomei-o como “SalárioVendedor”.
- Clique no Form1, e altere as seguintes propriedades:
  - Name = frmSalarioVendedor
  - Text = Calculando Salário Vendedor
- Em seguida insira os seguintes componentes seguindo o layout mostrado abaixo:
  - 4 TextBoxes
  - 5 Labels
  - 2 RadioButtons
  - 1 Button



- Altere as seguintes propriedades dos componentes:

Componente	Propriedade	Texto a ser inserido
Label1	NAME	lblNome
	TEXT	Nome
Label2	NAME	lblSalarioBruto
	TEXT	Salário Bruto
Label3	NAME	lblFuncao
	TEXT	Função
Label4	NAME	lblValorVendas
	TEXT	Valor de Vendas
Label4	NAME	lblSalarioReceber
	TEXT	Salário a Receber
TextBox1	NAME	txtNome
TextBox2	NAME	txtSalarioBruto
TextBox3	NAME	txtValorVenda
TextBox4	NAME	txtSalarioReceber
Button1	NAME	btnCalcular
	TEXT	Calcular
RadioButton1	TEXT	Vendedor
RadioButton2	TEXT	Outros

- Clique com o botão direito do mouse sobre o formulário e selecione a opção View Code. Em seguida declare as seguintes variáveis.

```
namespace SalarioVendedor
{
    publlic partial class frmSalarioVendedor : Form
    {
      string nome;
      double salarioBruto, valorVendas, salarioReceber;

      (...)
```

• Note que vamos pedir um Valor de Venda apenas se o usuário selecionou Vendedor no RadioButton. Para tanto vamos usar a propriedade ReadOnly do TextBox, conforme mostra o trecho de código abaixo:

```
private void frmSalarioVendedor_Load(object sender, EventArgs e)
{
    txtValorVenda.ReadOnly = true;
}

private void rdbVendedor_CheckedChange(object sender, EventArgs e)
{
    txtValorVenda.ReadOnly = false;
}

private void rdbOutros_CheckedChanged(object sender, EventArgs e)
{
    txtValorVenda.ReadOnly = true;
}
```

• Agora precisamos tornar possível o cálculo do salário através do evento click do botão calcular. Use o seguinte trecho de código:

```
private void btnCalcular_Click(object sender, EventArgs e)
{
    salarioBruto = System.Convert.ToDouble(txtSalarioBruto.Text);

    if (rdbVendedor.Checked == true)
    {
      valorVendas = System.Convert.ToDouble(txtValorVenda.Text);
      salarioReceber = ((valorVendas * 10) / 100) + salarioBruto;
      txtSalarioReceber = System.Convert.ToString(salarioReceber);
    }
}
```

• Analisando o código acima você poderá perceber que ao lidarmos com os valores informados nos TextBoxes precisamos primeiro convertê-los para um número em ponto flutuante através do uso do método System.Convert.ToDouble(). Isso ocorre porque a propriedade Text de um TextBox é do tipo String. Se quisermos fazer operações matemáticas com ela vamos precisar convertê-la para um número.

• Da mesma forma, quando quisemos mover o salário a receber para o outro TextBox, precisamos converter a variável salarioReceber (do tipo double) para uma String e só então colocar o valor resultante na propriedade Text.

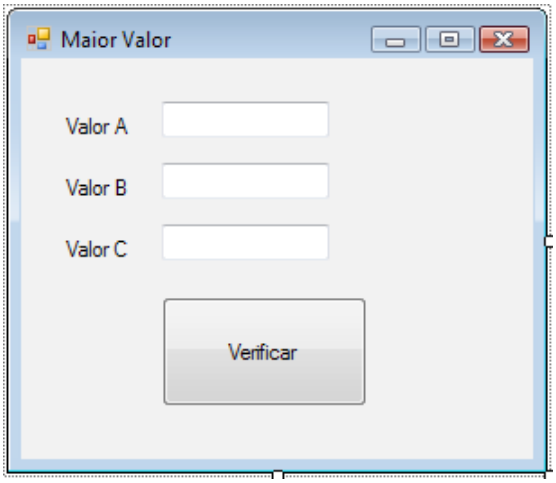
• Uma vez concluída a edição do código, salve o projeto e pressione F5 para executá-lo. Experimente entrar com alguns valores e verifique o resultado.

12.10 COMPARAÇÃO DE VALORES

O programa irá pedir que o usuário informe três valores e depois mostrar o maior deles.

Passos a serem seguidos:

- Crie um novo projeto WindowsForms e nomei-o “MaiorDeTres”.
- Selecione o Form1 e altere as suas propriedades conforme mostrado abaixo:
  - o NAME= “frmMaiorValorDeTres
  - o TEXT= “Maior Valor’
- Agora insira os seguintes componentes formulário, sempre seguindo o modelo mostrado abaixo:
  - o 4 Labels
  - o 3 TextBoxes
  - o 1 Button



- Altere as seguintes propriedades dos componentes

Componente	Propriedade	Texto a ser inserido
Label1	NAME	lblValorA
	TEXT	Valor A
Label2	NAME	lblValorB
	TEXT	ValorB
Label3	NAME	lblValorC
	TEXT	Valor C
Label4	NAME	lblResposta
	TEXT	(deixar vazio)
TexBox1	NAME	txtValorA
TexBox2	NAME	txtValorB

TexBox3	NAME	txtValorC
Button1	NAME	btnVerificar
	TEXT	Verificar

Faça a declaração das variáveis.
Insira no evento CLICK do botão o seguinte código:

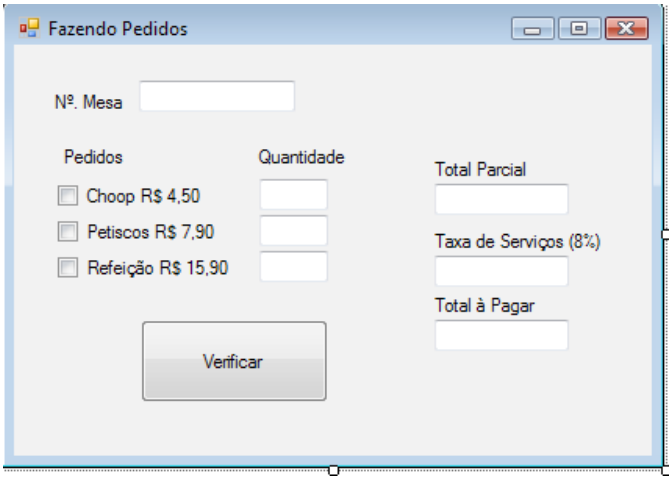
```
public partial class frmMaiorValorDeTres : Form
{
    public frmMaiorValorDeTres()
    {
      InitializeComponent();
    }

    private void btnVerificar_Click(object sender, EventArgs e)
    {
      int valorA = System.Convert.ToInt16(txtValorA.Text);
      int valorB = System.Convert.ToInt16(txtValorB.Text);
      int valorC = System.Convert.ToInt16(txtValorC.Text);

      if (valorA > valorB)
      {
            if (valorA > valorC)
            {
                  lblResposta.Text = "O Maior Valor é o A ";
            }
            else
            {
                  lblResposta.Text = "O Maior Valor é o C";
            }
      }
      else if (valorB > valorC)
      {
            lblResposta.Text = "O Maior Valor é o B";
      }
      else
      {
            lblResposta.Text = "O Maior Valor é o C";
      }
    }
}
```

12.11 CONTROLE DE PEDIDOS

Crie um programa onde o usuário possa fazer um mini-pedido na lanchonete, obtendo seu total e valor de serviços.

Siga os seguintes passos:

- Crie um novo projeto: “TestandoCheckBox”.
- Altere as seguinte propriedades do Form1:
  - o FILE = frmFazendoPedidos.cs
  - o NAME = frmFazendoPedidos
  - o TEXT = Fazendo Pedidos
- Insira os seguintes componente conforme o layout abaixo:
  - o 7 TextBoxes
  - o 6 Labels
  - o 3 CheckBoxes
  - o 1 Button



Altere as propriedades dos componentes, conforme tabela abaixo.

Componente	Prorpriedade	Texto a ser inserido
TextBox1	NAME	txtNumMesa
TextBox2	NAME	txtQtdChoop
TextBox3	NAME	txtQtdPetiscos
TextBox4	NAME	txtQtdRefeicao
TextBox5	NAME	txtTotalParcial
TextBox6	NAME	txtTaxaServico
TextBox7	NAME	txtTotalPagar
Label1	NAME	lblNumMesa
	TEXT	Nº. Mesa
Label2	NAME	lblPedidos
	TEXT	Pedidos
Label3	NAME	lblQuantidade
	TEXT	Quantidade
Label4	NAME	lblTotalParcial
	TEXT	Total Parcial
Label5	NAME	lblTaxaServico
	TEXT	Taxa de Serviço (8%)

Label6	NAME	lblTotalPagar
	TEXT	Total a Pagar
Button1	NAME	btnVerificar
CheckBox1	NAME	chkChoop
	TEXT	Choop R$ 4,90
CheckBox2	NAME	chkPetisco
	TEXT	Petisco R$ 7,90
CheckBox3	NAME	chkRefeicao
	TEXT	Refeição R$ 15,90

- Insira no evento click do botão o código marcado em negrito abaixo:

```
private void button1_Click(object sender, EventArgs e)
{
    double qtdChoop = Convert.ToDouble(txtQtdChoop.Text);
    double qtdPetisco = Convert.ToDouble(txtQtdPetisco.Text);
    double qtdRefeicao = Convert.ToDouble(txtQtdRefeicao.Text);

    double result1=0, result2=0, result3=0;

    if (chkChoop.Checked == true)
    {
      result1 = qtdChoop * 4.50;
    }
    if (chkPetisco.Checked == true)
    {
      result2 = qtdPetisco * 7.90;
    }
    if (chkRefeicao.Checked == true)
    {
      result3 = qtdRefeicao * 15.90;
    }

    double totalParcial = result1 + result2 + result3;
    txtTotalParcial.Text = Convert.ToString (totalParcial);

    double taxaServico = (totalParcial * 10) / 100;
    txtTaxaServico.Text = Convert.ToString (taxaServico);

    txtTotalPagar.Text = Convert.ToString(totalParcial + taxaServico);
}
```

Observe que foi necessário primeiramente fazer a conversão de dados obtidos através dos textboxes, assim atribuindo os respectivos valores às variáveis declaradas.

Logo em seguida declaramos as variáveis que irão receber valores dentro de uma estrutura de condição. Já que a condição do if pode não ser satisfeita, correríamos o risco de nunca inicializarmos algumas daquelas variáveis, então para evitar isso as inicializamos com 0 (zero) no momento da sua declaração.

13 TRATAMENTO ESTRUTURADO DE EXCEÇÕES

Quando um programa está em execução pode surgir a necessidade de tratar erros que ocorrem por motivos inesperados, como por exemplo a divisão de um número por zero ou a tentativa de abrir um arquivo que não existe. Essas são apenas algumas situações que podem ocorrer e que não podemos evitar. Podemos, sim, criar código específico para informar ao usuário que a operação que ele pretende realizar é inviável. Um erro deste tipo, se não tratado, pode encerrar o programa de forma inesperada ou gerar uma mensagem incompreensível para o usuário.

As instruções do C# utilizadas para tratamento de exceções estruturadas são try, catch, finally e throw:

Instrução	Descrição
try	Contém código onde pode ocorrer uma ou mais exceções, como por exemplo a tentativa de abrir um arquivo inexistente, ou uma tentativa de dividir um número por zero.
catch	Contém o código usado para tratar um determinado tipo de erro.
finally	É sempre executado. É útil para liberar recursos usados dentro do bloco try, como, por exemplo, fechar uma conexão com um banco de dados, fechar um arquivo texto, etc.
throw	Dispara uma exceção.

13.1 INSTRUÇÕES TRY E CATCH

O bloco try contém o código onde o erro pode ocorrer, o bloco catch, o código para manipular o erro ocorrido.

```
try
{
    // Código que pode lançar uma exceção.
}
catch (<tipo-de-exceção> <variável-de-exceção>)
{
    // Código.
}
```

Veja um exemplo:

```
try
{
    double n = 5 / 0;
}
catch (System.DivideByZeroException e)
{
    System.Console.WriteLine(“Tentamos dividir por zero.”);
    System.Console.WriteLine(“Um erro foi lançado.”);
}
```

13.2 ALGUMAS EXCEÇÕES COMUNS

Algumas exceções são lançadas automaticamente pelo Common Language Runtime quando as operações básicas falha. Estas exceções e suas causas de erro são mostradas na tabela abaixo:

Exceção	Causa
System.ApplicationException	Lançada quando ocorre um erro não-fatal de aplicação.
System.ArgumentException	Lançada quando um argumento para um método não é válido.
System.ArithmeticException	Uma classe base para exceções que ocorrem durante uma operação aritimética, como por exemplo DivideByZeroException e OverflowException
System.ArrayTypeMismatchException	Lançada quando um array não pode armazenar um dado elemento porque o tipo de dados do elemento é incompatível com o tipo de dados do array.
System.DivideByZeroException	Lançada quando ocorre uma tentativa de dividir um valor integral por zero.
System.FormatException	Lançada quando o formato de um argumento não corresponde ao parâmetro especificado pelo método invocado.
System.IndexOutOfRangeException	Lançada quando ocorre uma tentativa de acessar um elemento de um array quando o índice é menor do que zero ou é maior do que o tamanho do array.
System.InvalidCastException	Lançada quando uma conversão explícita de um tipo de dados base para uma interface ou um tipo de dados derivado falha em tempo de execução.
System.NotFiniteNumberException	Ocorre quando uma operação de ponto flutuante resulta em infinito (seja positivo ou negativo) ou em um NaN (Not a Number).
System.NullReferenceException	Ocorre quando você tenta acessar um objeto cujo valor corrente é nulo.
System.OutOfMemoryException	Lançada quando uma tentativa de alocar memória usando o operador new falha. Isso indica que a memória disponível para o CLR foi completamente exaurida.
System.OverflowException	Lançada quando uma operação aritimética gera um valor maior do que o maior valor possível de se armazenar em um dado tipo de dados.
System.StackOverflowException	Lançada quando a pilha de execução é estourada por se ter muitas chamadas de métodos pendentes. Isso normalmente indica uma recursão profunda demais ou mesmo infinita.
System.TypeInitializationException	Lançada quando um construtor estático lança uma exceção e não existe um catch compatível para capturá-la.

13.3 MÚLTIPLOS BLOCOS CATCH

Você também pode capturar múltiplas exceções diferentes ao tentar tratar um erro. Mas tome cuidado: ao usar múltiplas exceções, trate primeiro aquelas de com profundidade maior, para só depois tratar as exceções mais genéricas.

Por exemplo, como mostrado na tabela acima, DivideByZeroException e OverflowException estendem ArithmeticException, então, ao tentar tratá-las, coloque primeiro as subclasses para depois tratar a superclasse. Veja o exemplo.

```
try
{
    // Código que pode lançar uma exceção aritmética
}
catch (DivideByZeroException dbze)
{
    // Código para tratar uma divisão por zero.
}
catch (OverflowException oe)
{
    // Código para tartar um erro de overflow.
}
catch (ArithmeticException ae)
{
    // Código para tratar alguma outra exceção aritmética.
}
catch (Exception e)
{
    // Código para tratar uma exceção qualquer.
}
```

13.4 INSTRUÇÃO THROW

A instrução throw permite disparar uma exceção. Ela é utilizada freqüentemente para simular situações de erro, ou quando você quer lançar suas próprias exceções. Isso evita que tenhamos de testar fisicamente todas as condições que causariam erros. Por exemplo, as linhas de código a seguir simulam um tratamento de erro exatamente como se um erro de divisão por zero tivesse ocorrido.

```
Try
{
    throw new DivideByZeroException();
}
catch (Exception ex)
{
    // Código que captura o erro.
}
```

Esse código parece não fazer muito sentido (porque iríamos querer lançar um erro para simplesmente capturá-lo logo depois. Mas acredite, ele é útil. Não na forma simples em que mostramos, mas podemos usar uma construção semelhante para lançarmos nossas próprias exceções. Imagine, por exemplo, que você está criando uma aplicação financeira e quer notificar ao usuário quando ele tenta fazer uma retirada maior do que o seu limite permite: é exatamente nestas situações que vamos querer lançar nossas próprias exceções – logicamente a operação poderia funcionar (ou seja, nenhuma exceção de sistema seria lançada, já que o registro do cliente existe, a operação matemática que geraria o novo saldo não causa erros e mesmo a atualização no banco poderia ser feita sem problemas, mas do ponto de vista da lógica de negócios aquela operação não deve ser permitida.

13.5 INSTRUÇÃO FINALLY

Esta instrução é sempre executada independente do fato de ocorrer ou não uma exceção. É útil para fechar uma conexão com um banco de dados, arquivo-texto etc.

```
Connection conn;
try
{
    // Obtém uma conexão com o banco e faz algum uso útil
    // dela.  Este código pode lançar uma exceção.
}
catch (Exception e)
{
    // Código de tratamento da exceção.
}
finally
{
    // Código que fecha a conexão com o banco, independente
    // de ter ocorrido um erro ou não.
    if (conn != null)
    {
      conn.close();
    }
}
```

14 ORIENTAÇÃO A OBJETOS

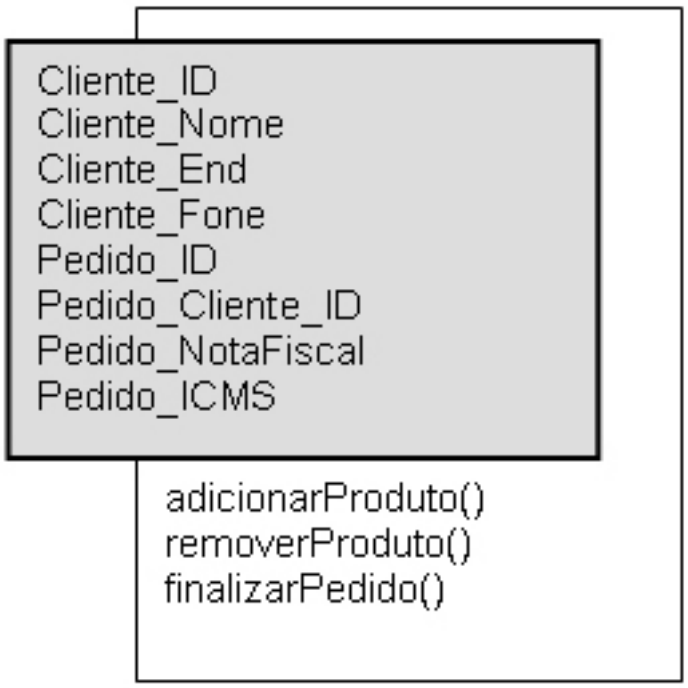
14.1 INTRODUÇÃO

Até há alguns anos atrás, o desenvolvimento de software baseou-se no chamado modelo procedural. Mas, depois de muitos estudos, a comunidade de Tecnologia da Informação desenvolveu uma técnica mais interessante, simples e natural para o processo de análise de problemas e desenvolvimentos de aplicações: a Orientação a Objetos (OOP- Object Oriented Programming).

Para compreender mais o significado dessa mudança, e o porquê de uma massiva migração de profissionais do modelo procedural, para a orientação a objetos. Vamos primeiramente fazer uma comparação das duas formas de programar.

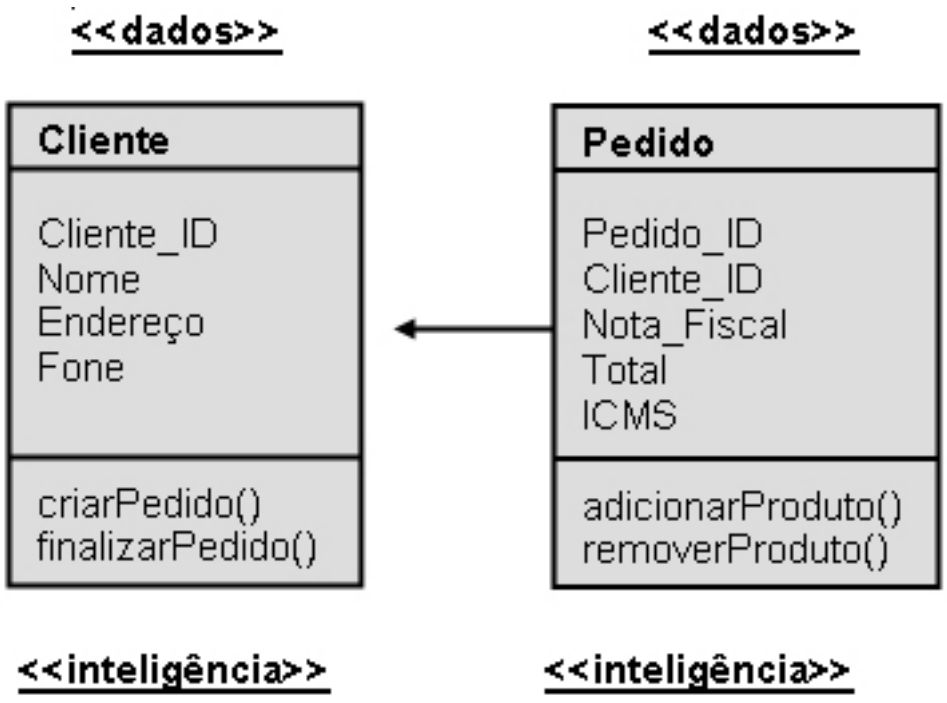
14.2 MODELO PROCEDURAL

Neste modelo, dados e programa encontram-se em estruturas computacionais diferentes. Essa característica dificulta a reunião desses elementos em unidades, fazendo com que os sistemas resultantes sejam muito complexos.



14.3 MODELO ORIENTADO A OBJETOS

A organização dos elementos: funções e dados ficam numa mesma área de memória, em uma mesma estrutura computacional. Assim, a construção de sistemas torna-se mais simples.



Na programação procedural você descreve os passos que tomaria para solucionar um problema, enquanto que na programação orientada a objetos você descreve os objetos que compõem esta solução e a interação entre eles.

Por causa das diferenças estruturais e conceituais, o modelo orientado a objetos é mais natural, mais próximo de nossa compreensão da realidade. De um ponto de vista prático, portanto, a orientação a objetos permite a redução de custos na produção de softwares e um melhor entendimento sobre o sistema desenvolvido.

Atualmente, existem várias linguagens de programação que permitem a aplicação total das técnicas da orientação a objetos. São elas: Java, C++, SmalTalk, .NET e etc.,

14.4 DIFERENÇAS ENTRE CLASSES E OBJETOS

Todo mundo fala em Programação Orientada a Objetos, mas o primeiro conceito que vimos foi Classe, e não Objeto. Por que isso?

Em C# você programa usando classes, e estas classes mais tarde gerarão objetos quando o programa estiver sendo executado. Grosso modo, uma classe é um conceito, um modelo para um objeto. Quando programa você define as classes, e elas ganham vida através dos objetos gerados a partir delas.

Normalmente você não usará as classes diretamente em seu programa. Ao invés disso, você as usaria apenas para instanciar novos objetos.

Imagine uma classe como uma planta de uma casa: você não pode morar na planta da casa, mas pode usá-la para criar quantas casas quiser.

14.5 CLASSES

As Classes nos permitem organizar de forma coerente o código que criamos. Por exemplo, podemos criar uma classe chamada Custo que contém todas as funções necessárias para manipular os cálculos referentes aos custos de uma empresa. Quando criamos uma aplicação, iremos criar diversas classes, cada uma representando um componente do problema. É a interação entre estes diferentes componentes que gera os resultados esperados pela aplicação.

Diferente do que acontecia na metodologia procedural, na OOP dividimos nosso programa em classes por área de interesse. Por exemplo, para modelar uma aplicação de videolocadora iríamos criar várias classes (uma para filme, outra para

ator, outra para produtora, outra para cliente, e assim por diante) ao invés de colocarmos todo o código em um único arquivo fonte.

Além de ajudar na compreensão do código, isso facilita sobremaneira a manutenção do mesmo: você pode se sentir tentado a comparar uma classe a um programa da metodologia procedural. A comparação até faz um certo sentido no início, mas à medida que você for ganhando experiência perceberá que as classes são muito mais poderosas do que isso. Se for bem projetada, uma classe poderá ser usada sem qualquer modificação em uma outra aplicação. E, ao contrário do que normalmente acontece na programação procedural, você não precisará copiar o código de uma classe para outro projeto quando quiser utilizá-la: basta fazer referência a ela e o run-time cuida do resto para você.

Talvez ainda não tenha ficado claro, então vale a pena reforçar: uma classe é um tipo de dados, o tipo de dados mais poderoso do C#. Classes definem dados e comportamentos em uma estrutura única. Você modela a classe no seu código fonte e, a partir dele, cria quantos **Objetos** daquele tipo precisar.

Como dissemos antes, uma classe comporta dados e comportamentos, os chamados membros da classe. Na notação de OOP chamamos os dados de atributos, e comportamentos são métodos.

Para acessar os membros de uma classe, devemos criar uma instância da classe, um objeto cujo tipo de dados é a classe, ou seja, um objeto é uma entidade concreta baseada em uma classe (uma entidade abstrata).

Atributos: representam os dados associados à uma **instância** de uma classe.

Métodos: podem ser comparadas aos procedimentos e funções que existiam na metodologia procedural. Estes métodos atuarão sobre os atributos para gerar os resultados esperados pelo programador.

Uma classe é declarada com a instrução class e pode ser precedida de um modificador de acesso. Sintaxe:

```
[modificadores] class <nome-da-classe>
{
    <atributos>
    <métodos>
}
```

Exemplo:

```
public class Taxa
{
    // Código da classe.
}
```

Os objetos podem conter dados e ter comportamento: os dados dos objetos são armazenados em campos, propriedades, e eventos do objeto, e o comportamento dos objetos é definido pelos métodos e interfaces do objeto.

Algumas características dos objetos em C#.

- Tudo que vocês usa em C# é um objeto, inclusive formulários Windows (Windows Forms) e controles.
- São definidos como modelos de classes
- Usam propriedades para obter e modificar as informações que contêm.
- Têm métodos e eventos que permitem executar ações.
- Todas as classes herdam da classe System.Object.

Cada objeto precisa ser uma instância de uma classe, e são criados por meio da palavra-chave new.

```
<nome-da-classe> <nome-da-instância> = new <nome-da-classe>( );
```

Para criar uma instância de uma classe hipotética chamada Taxa usaríamos a seguinte sintaxe:

```
Taxa objetoTaxa = new Taxa();
```

Quando uma instância de uma classe é criada, uma referência para o objeto é passada para o programador. No exemplo anterior, objetoTaxa é uma referência para um objeto baseado na classe Taxa. A classe define um tipo de objeto, mas não é um objeto. Neste caso, podemos entender objetoTaxa como uma variável do tipo Taxa, exatamente como se tivéssemos criado uma variável idade do tipo int:

Quando uma instância de uma classe é criada, uma referência para o objeto é passada para o programador. No exemplo anterior, objetoTaxa é uma referência para um objeto baseado na classe Taxa. A classe define um tipo de objeto, mas não é um objeto. Neste caso, podemos entender objetoTaxa como uma variável do tipo Taxa, exatamente como se tivéssemos criado uma variável idade do tipo int:

```
int idade = 23;
```

A diferença, talvez nem tão evidente, é o uso da palavra-reservada new. Ela só é usada para instanciar novos objetos.

14.6 MODIFICADORES DE ACESSO PARA CLASSES

Os modificadores de acesso são utilizados para restringir o acesso às classes e a seus membros (atributos e métodos).

Os modificadores de acesso de classe permitidos são:

Modificadores	Descrição
public	A classe é acessível em qualquer lugar. Não há restrições.
private	Acessível apenas dentro do contexto em que foi declarada.
protected	A classe é acessível somente na própria classe e nas classes derivadas.

Todo o tipo declarado dentro de uma classe sem um modificador de acesso é considerado, por padrão, como private.

Os modificadores de acesso protected e private só são permitidos em classes aninhadas. Veja o exemplo:

```
public class Taxas
{
    // Código da classe Taxas.
    private class Juros
    {
      // Código da classe Juros.
    }
}
```

Os tipos aninhados são private por padrão e podem acessar membros definidos como private e protected da classe em que estão contidos. Devemos usar o nome qualificado completo para acessar os membros de classes aninhadas.

Para criar uma instância de classe aninhada, usamos:

```
public static class Livro
{
    // Código da classe estática.
    // Membros declarados aqui serão estáticos também.
}
```

As classes estáticas são carregadas automaticamente pelo CLR quando o programa que contém a classe é carregado. Algumas características das classes estáticas:

- Contêm somente membros estáticos.
- Não é possivel criar uma instância.
- Não podem ser implementadas por intermédio de herança.
- Não podem conter um método construtor.

Membros estáticos são úteis em situações em que o membro não precisa estar associado a uma instância da classe, como, por exemplo, em situações que o valor contido no membro não se altera, como é o caso do nome do autor contido no campo

Autor.

```
public static class Universo {
    public static int velocidadeDaLuzEmKmPorSegundo = 300000;
}
```

Para acessar o campo velocidadeDaLuzEmKmPorSegundo, simplesmente usamos o nome da classe, seguido do ponto e do nome do campo:

```
Universo.velocidadeDaLuzEmKmPorSegundo;
```

Exemplo completo:

```
public static class Universo
{
    public static int velocidadeDaLuzEmKmPorSegundo = 300000;
}

class TesteUniverso
{
    static void Main()
    {
      System.Console.WriteLine(Universo. velocidadeDaLuzEmKmPorSegundo);
    }
}
```

14.7 MÉTODOS

Um método contém uma série de instruções. Os métodos são procedimentos que têm a função de executar as tarefas programadas dentro das classes.

Os métodos devem ser declarados dentro de classes especificando o nível de acesso, o valor de retorno, o nome de método e parâmetros (opcional). Os parâmetros devem estar dentro de parênteses e separados por vírgula. Parênteses vazios indicam que o método não requer nenhum parâmetro. Sintaxe utilizada pelos métodos em C#.

```
[modificadores-de-acesso] <tipo-de-dados-de-retorno> <nome-do-método>([<tipo-de-dados-do-parâmetro-1> <nome-do-parâmetro-1>[, <tipo-de-dados-do-parâmetro-2> <nome-do-parâmetro-2>[, <tipo-de-dados-do-parâmetro-3> <nome-do-parâmetro-3>[, ...]]]])
```

Exemplo:

```
class Taxas
{
    public void Imprimir()
    {
    }
    public int Calculo(int x, int y)]
    {
      return 0;
    }
    public string Mensagem(string msg)
    {
      return msg;
    }
}
```

14.8 MODIFICADORES DE ACESSO PARA MÉTODOS

Os modificadores de acesso dos métodos são os mesmos utilizados com as classes. Quando definimos a acessibilidade de um método, devemos ter em mente que um membro de uma classe não pode ter mais privilégios que a classe que a contém.

Exemplo: se uma classe for declarada com o modificador private, todos os seus membros serão private.

Métodos declarados dentro de uma classe sem um modificador de acesso são considerados, por padrão, com private. Confira os modificadores de métodos do C#.

Modificadores	Descrição
public	O método é acessível em qualquer lugar. Não há restrições.
private	Acessível apenas dentro do contexto em que foi declarado.
protected	O método é acessível somente na própria classe e nas classes derivadas.

Chamamos um método por intermédio da referência ao objeto criado a partir da classe que contém o método, seguido de um ponto, o nome do método e parênteses. Os argumentos, se houver, são listados dentro dos parênteses e separados por vírgula. No exemplo, utilizamos novamente a classe Taxas que contém os métodos Imprimir, Calculo e Mensagem.

Exemplo:

```
Taxas objTax = new Taxas();
objTax.Imprimir();
objTax.Calculo(10, 20);
objTax.Mensagem("Olá Mundo");
```

A palavra-chave return pode ser utilizada para parar a execução de um método. Se ela não aparecer, a execução é encerrada somente no final do bloco de código.

Métodos que contêm um valor de retorno diferente de void devem obrigatoriamente utilizar a palavra-chave return indicando o valor a ser retornado.

Exemplo 1 – retorna uma string

```
Taxas objTax = new Taxas();
objTax.Imprimir();
objTax.Calculo(10, 20);
objTax.Mensagem("Olá Mundo");
```

Exemplo 2 – interrompe a execução do método se o parâmetro str for nulo:

```
public string Mensagem(string str)
{
    if (str == null)
    {
      return "";
    }
    else
    {
      return str;
    }
}
```

14.9 PARÂMETROS

Uma das grandes vantagens dos métodos é que eles nos permitem criar subrotinas, pequenos trechos de programa que se repetem. Ao invés de escrever o código que verifica se um CPF é válido toda a vez que esta operação é necessária, por exemplo, poderíamos criar um método validaCPF que executa a tarefa, e o chamaríamos toda a vez que necessário.

Mas isso só se torna uma vantagem efetiva se pudermos passar parâmetros para o método, ou seja, se pudermos identificar que CPF validar (ao invés de validarmos sempre o mesmo CPF, o que não faria o menor sentido ...).

Parâmetros, então, são pequenas incógnitas no código de um método, incógnitas que serão supridas por quem chamou o método. Quando declaramos um método que recebe parâmetros devemos especificar o tipo de dados que o parâmetro pode conter.

```
public void Metodo(int x, int y)
{
    System.Console.WriteLine("X: " + x);
    System.Console.WriteLine("Y: " + y);
}
```

O método, então, poderia usar os parâmetros recebidos para customizar a tarefa a ser executada.

14.10 TIPO DE RETORNO

Um método é na verdade uma função, daquelas que aprendemos com as equações na quinta série. Uma função pode receber um parâmetro e também pode retornar um valor. Um método pode, por exemplo, receber dois valores e retornar o primeiro elevado à potência do segundo.

Ou, no caso do exemplo da validação de CPF, retornar um valor booleano que indica se o número é válido ou não. Veja o exemplo.

```
public boolean validaCPF(string cpf)
{
    // Faz a mágica, e verifica se o CPF é realmente válido.
    if (/* Qualquer que seja o teste ...*/)
    {
      return true;
    }

    // Se chegou até aqui, o CPF não é válido.
    return false;
}
```

14.11 SOBRECARGA DE MÉTODOS

Cada membro usado por uma classe possui uma assinatura única. A assinatura de um método consiste do nome do método, e a sua lista de parâmetros.

Podemos declarar numa classe vários métodos com o mesmo nome mas com parêmetros diferentes. Isso é chamado de sobrecarga de métodos, cuja principal vantagem é a flexibilidade. Um único método, por exemplo, pode ter várias implementações, dependendo do tipo e número de parâmetros que recebe.

Veja a seguir o exemplo completo possui quatro implementações do método teste.

```
class Livro
{
    public void teste()
    {
      System.Console.WriteLine("Método teste()");
    }
    public int teste(int i)
    {
      return i;
    }
    public string teste(int a, string str)
    {
      return str + "" + a;
    }
    public string teste(string str, int a)
    {
      return a + "" + str;
    }
}

class TesteLivro
{
    static void Main()
    {
      Livro lv = new Livro();
      lv.teste();

      Console.WriteLine(lv.teste(8));
      Console.WriteLine(lv.teste(10, "Segunda string"));
      Console.WriteLine(lv.teste("Primeira string", 10));
    }
}
```

O exemplo retorna os valores passados a cada implementação do método teste.

```
Método teste()
8
Segunda string 10
10 Primeira string
```

Preste atenção nas seguintes regras ao usar sobrecarga de métodos:

- Todos os métodos sobrecarregados devem ter o mesmo nome.
- Os métodos devem ser diferentes entre si em pelo menos um dos seguintes itens:

o Número de parâmetros;
o Ordem dos parâmetros;
o Tipo de dados dos parêmetros;
o O tipo de retorno (válido somente em conversões de operadores).

14.12 CONSTRUTORES

Sempre que uma classe é criada um método especial é chamado. É o seu método construtor.. Um método construtor possiu o mesmo nome da classe em que está deckaradi e não retorna valor, nem mesmo void.

Uma classe pode conter múltiplos construtores com diferentes argumentos. Um construtor sem parâmetros é chamado de construtor-padrão.

Todas as classes possuem um construtor. Se você não declarar um construtor o compilador C# criará automaticamente um construtor-padrão para você.

Usando construtores, um programador pode estabelecer valores-padrão ou definir os valores iniciais dos membros. Veja o exemplo:

```
public class Livro
{
    public string titulo;
    public string autor;

    public Livro()
    {
      titulo = "ASP .NET com C#";
      autor = "Alfredo Lotar";
    }
}
```

No exemplo acima, a definição dos valores iniciais dos campos titulo e autor, acontece no construtor da classe Livro.

Podemos reescrever o exemplo com novos construtores, usando sobrecarga de métodos. Veja o exemp

```
public class Livro
{
    public string titulo;
    public string autor;

    public Livro()
    {
      titulo = “ASP .NET com C#”;
      autor = “Alfredo Lotar”;
    }

    public Livro(string tit, string aut)
    {
      titulo = tit;
      autor = aut;
    }
}
```

Criando instância da classe Livro nos dois casos:

Construtor sem parâmetros

```
static void Main(string[] args)
{
    Livro umLivro = new Livro();
    Livro outroLivro = new Livro(“C# Passo a Passo”, “Microsoft”);
}
```

:

15 OS PRINCÍPIOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

15.1 CLASSE

É uma estrutura para incorporar os dados e os procedimentos para trabalhar com esses dados. Por exemplo, se você estivesse interessado em controlar os dados associados aos produtos no inventário de uma empresa, criaria uma classe Produto responsável por manter e trabalhar com os dados pertinentes aos produtos. Se quisesse ter capacidades de impressão em sua aplicação, trabalharia com uma classe Impressora, responsável pelas funcionalidades que exigem a interação com uma impressora do mundo real.

As características mais importantes de uma classe é que ela se compõe basicamente de dois elementos, dados e comportamento, e que tenta modela um objeto da vida real usando conceitos como abstração e encapsulamento.

15.2 ABSTRAÇÃO

É a capacidade de selecionar os atributos que são relevantes para determinada situação. Por exemplo: quando dirijo meu carro, preciso saber a velocidade do carro e a direção na qual ele está indo. Como boa parte da mecânica do carro é automática, não preciso saber a quantas RPMs está o motor, ou se a baixa temperatura está fazendo o carro consumir mais, portanto, filtro essas informações. Por outro lado, essas informações seriam críticas para um piloto de carros de corrida, que não as filtraria.

15.3 ENCAPSULAMENTO

Consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados. Exemplo: para usar o aparelho de telefone, não é necessário conhecimento dos circuitos internos do aparelho ou como a rede telefônica funciona, basta saber manipular os botões e o fone.

15.4 POLIMORFISMO

Diferentes implementações de uma mesma função podem ser usadas conforme as condições atuais (ou conforme o tipo de dados que você fornece como argumento). Exemplo: nos carros que usam o controle de tração, acelerar() pode se comportar de maneira diferente conforme o tipo ou as condições do terreno.

15.5 HERANÇA

Você usará a herança na OOP para classificar os objetos em seus programas, de acordo com as características comuns e a função. Isso torna mais fácil e claro trabalhar com os objetos. Também torna a programação mais fácil, porque permite que você combine as características gerais em um objeto-pai e herde essas características nos objetos-filhos. Por exemplo, poderá definir um objeto funcionário, que define todas as características gerais do funcionário, mas também adiciona características exclusivas aos gerentes em sua empresa. O objeto gerente refletirá automaticamente qualquer alteração na implementação do objeto funcionário.