

Capítulo II – A Linguagem de Programação C#

Introdução

A ECMA é uma organização que foi fundada em 1961 e dedica-se à padronização dos Sistemas de Informação. Após 1994, passou a ser conhecido com ECMA International para refletir mais amplamente o seu escopo internacional. Em 2000, o ECMA recebeu oficialmente as especificações da linguagem C#, um conjunto de bibliotecas de classes e o ambiente para a execução da padronização (a plataforma .NET em si). As especificações de código detalhando a C#, as bibliotecas de classes e funções internas do CLR foram disponibilizadas gratuitamente não sendo componentes proprietárias.

Isto permite que você possa construir a sua própria IDE para esta linguagem e aliás, já existe uma gratuita muito boa, a **Sharp Develop** (baixaki, encontra-se no). O trabalho de padronização foi elaborado pelo comitê ECMA TC39, o mesmo que padronizou a linguagem JavaScript.

Estrutura e especificação da linguagem C#

A nossa pequena monografia não pretende estabelecer todas os itens e características da linguagem C#, ela é extremamente complexa e demandaria um livro de no mínimo algumas centenas de páginas para arranharmos um pouco a sua superfície, então vamos falar aqui apenas dos componentes mais importantes da linguagem: classes e objetos.

Uma classe é uma estrutura de dados ativa. Antes dos dias da programação orientada a objetos, programadores organizavam os programas como uma seqüência de instruções, compiladas num ambiente de alto nível. Com o advento da programação orientada a objetos (OOP), os programas são hoje em dia encarados como estruturas de dados e funções organizados em conjuntos logicamente coerentes e encapsulados, de modo a favorecerem reutilização e componentização, manutenção e versioning (controle de versão). Os itens de dados são organizados em conjuntos denominados classes, e as instâncias das classes são objetos e métodos (ações sobre objetos). Uma classe é uma estrutura que pode armazenar dados e executar código através de métodos sobre os seus objetos.

A figura abaixo mostra a estrutura geral de um programa C# da plataforma .NET.

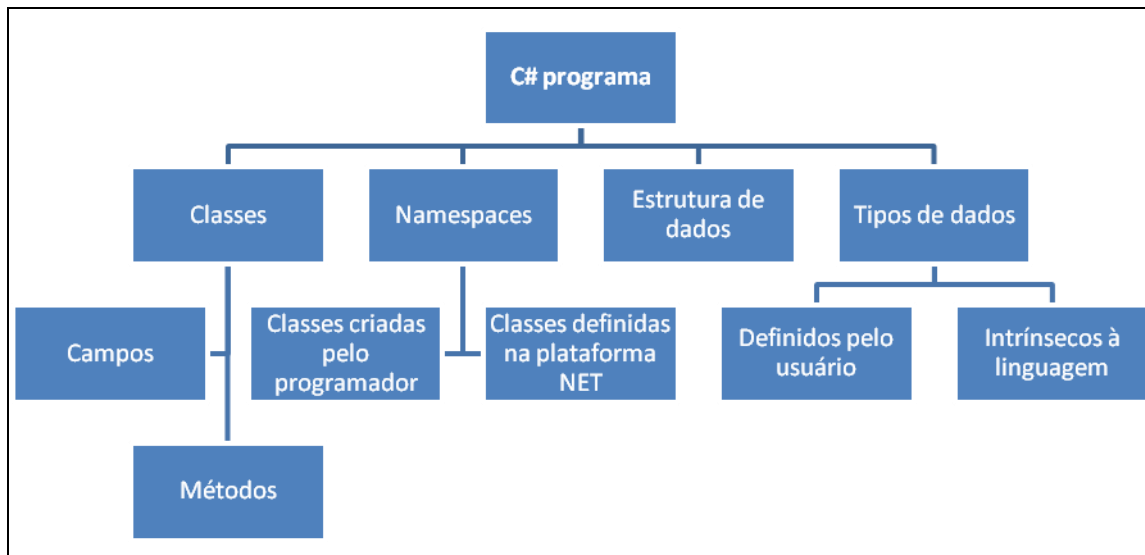


figura 4: Um programa C# e suas principais estruturas.

Classes, métodos e campos em C#

Campos e métodos são os principais componentes de uma classe. Campos são membros de dados e métodos são membros de funções.

Um campo é uma variável que pertence a uma classe, podendo ser de 2 tipos:

- Pré-definida
- Definida pelo usuário

Campos armazenam dados, portanto constituem as estruturas de dados em si.

Vamos criar um novo projeto no **Visual C# 2008 Express**, do tipo aplicação console. Note que ao abrirmos **New Project**, o Visual C# disponibiliza os templates (gabaritos) disponíveis no ambiente atual. Ele pode ser customizado (aperfeiçoado), conforme novos plugins e templates são disponibilizados pela Microsoft.

Três tipos de aplicações para aprendermos C#: Console, WEB Browser e Bancos de Dados.

Há diversos tipos de projeto que a plataforma pode construir, como discutimos nas seções acima, no entanto, o novato em C# (e principalmente se for novato na programação em geral) não deve subestimar o que pode ser aprendido com as aplicações **Console**, e estrategicamente falando, o programador em evolução acaba percebendo que todas as tecnologias da OO e comunicações de dados também são suportadas pelas aplicações **Console**. Aplicações que envolvem polinômios e matrizes podem ser executadas perfeitamente neste ambiente não-gráfico.

Em seguida, o estudante de programação deve dar os seus passos em direção aos aplicativos **Windows Forms** (apenas quando tiver dominado algoritmos razoáveis feitos em Console) e construir os seus gabaritos (templates) de classe.

Então vamos começar o nosso estudo das características da linguagem através de alguns projetos **Console**.

Aplicações Console – iniciando o conhecimento da OOP

Inicie o programa **Microsoft Visual C# 2008 Express Edition**, se o mesmo não já estiver aberto, e clique em **File**. A seguinte tela no Windows deve aparecer:

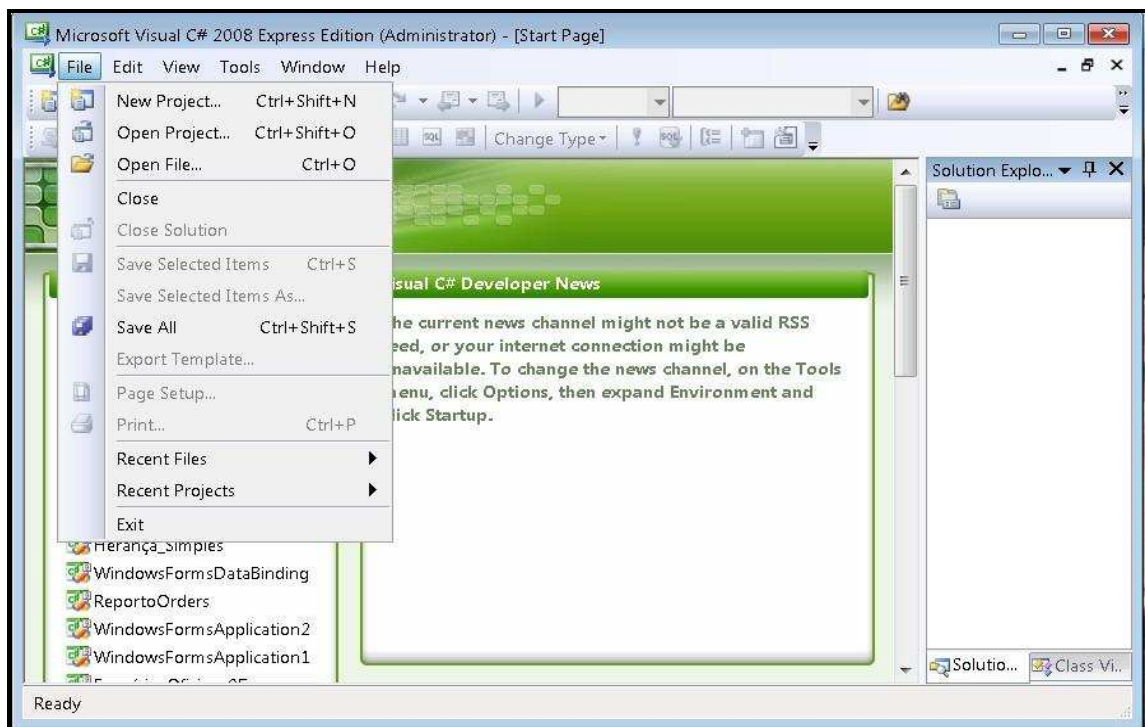


figura 5: abrindo o Visual C# 2008 e um novo projeto.

Ao selecionarmos File -> New Project, a seguinte caixa de diálogo abaixo é aberta:

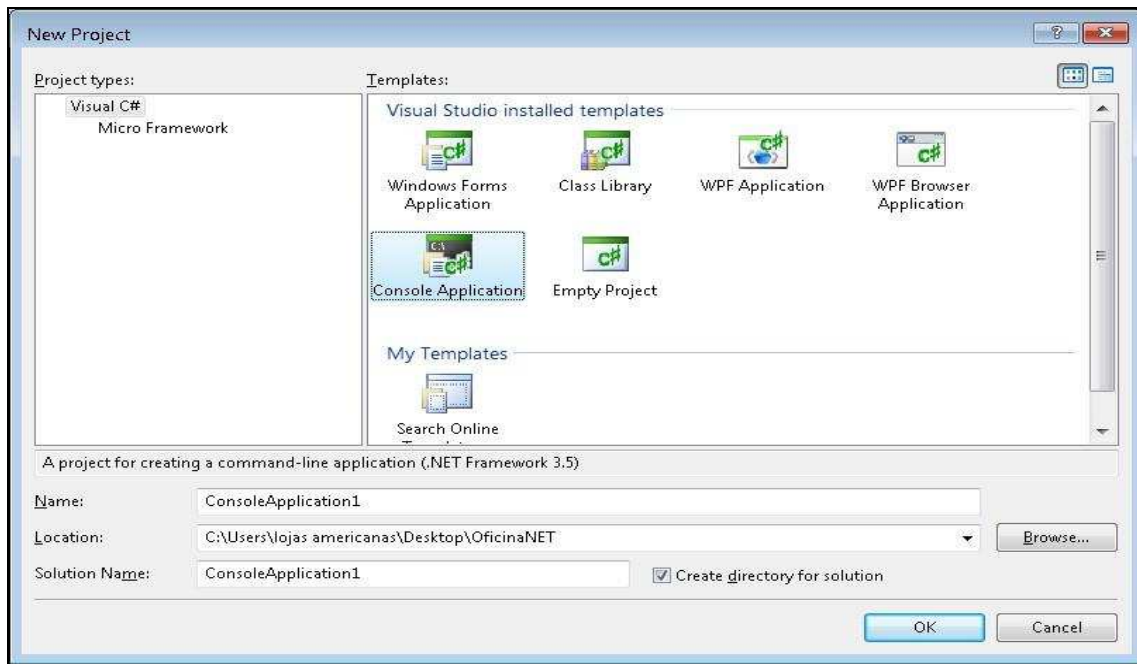


figura 6: criando um novo projeto no Visual C# 2008

Após escrevermos o nome do projeto na caixa de diálogo acima, em **Name**, configure Location e Solution Name para registrarmos onde guardaremos os novos futuros arquivos de projetos. Marque a caixa **Create directory for solution** e dê OK. A seguinte tela deve aparecer:

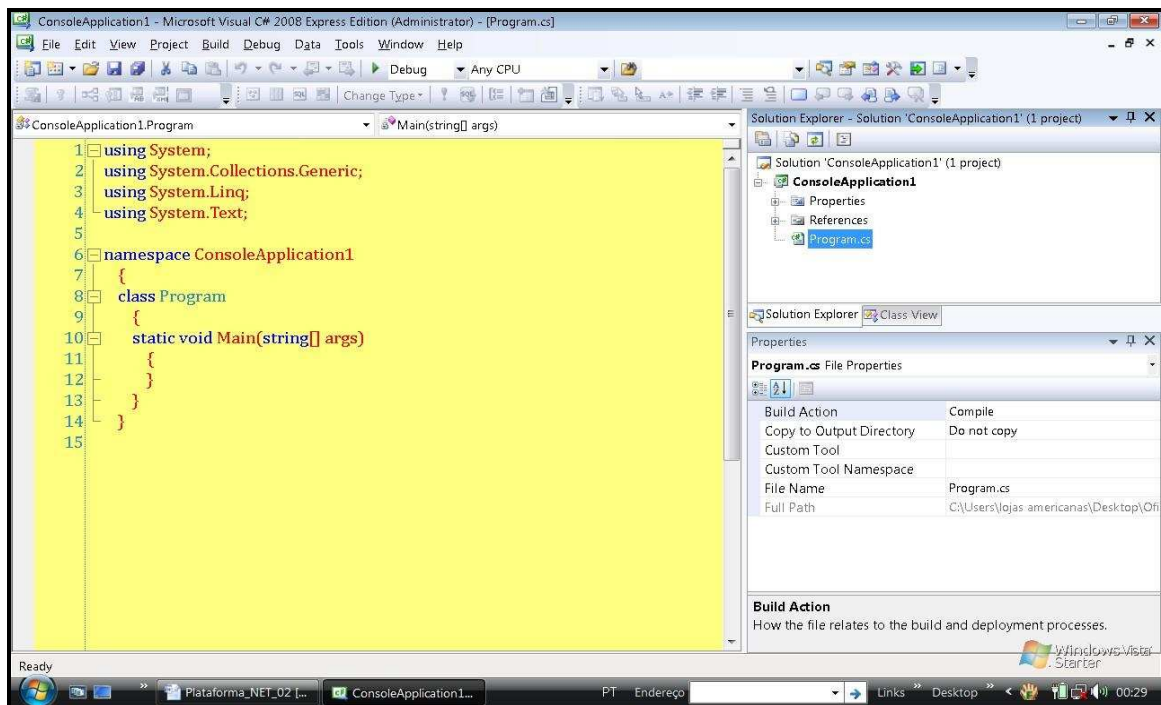


figura 7: o código de Main, contido em Program.cs.

O nosso projeto vê-se dividido em três grandes setores: a área de **edição de código** (em amarelo), o solution explorer e o painel properties que mostra todas as propriedades de

algum objeto selecionado. Neste caso selecionei Program.cs (que é um arquivo-fonte da linguagem C#).

A classe **Program** possui um método **Main**, responsável pelo carregamento e execução do programa principal. Por default, esta classe é denominada **Program**, mas seu nome pode ser mudado neste momento. Antes disso, escrevemos `Console.ReadLine()`; ao final do **Main**, com o objetivo de segurar a tela.

Na verdade, este método aguarda a entrada de string pelo usuário. No solution explorer, no canto superior à direita, podemos clicar no nome da classe (Program.cs) e com o botão direito do mouse solicitar a renomeação da classe. Cuidado para não usar palavras reservadas como `Console` e não se esqueça da terminação `.cs`, que identifica este arquivo como arquivo tipo C#, pronto para ser compilado pela IDE.

Digite **Console.ReadLine()**; no método **Main** da classe **Program**:

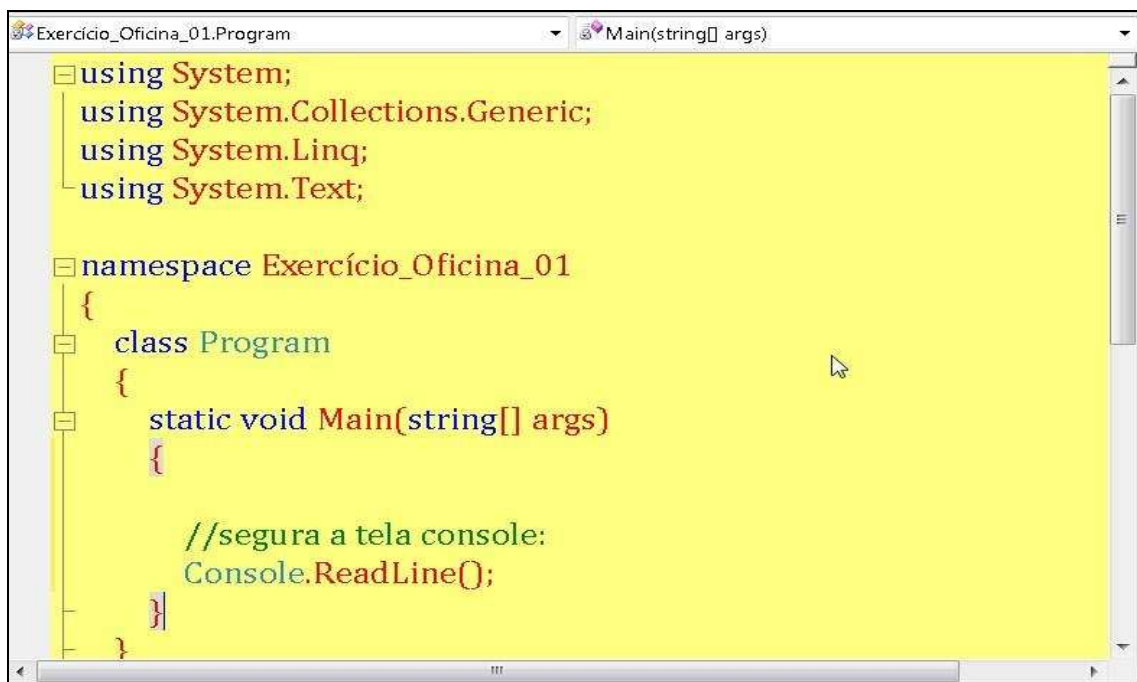


figura 8: um arquivo .cs com a principal classe do programa tipo console

Note que o namespace contém as classes do programa. Ao mudarmos o nome da classe, e clicarmos **Build**, o output é a janela que mostra erros de compilação. Geralmente aparece na porção inferior da tela:

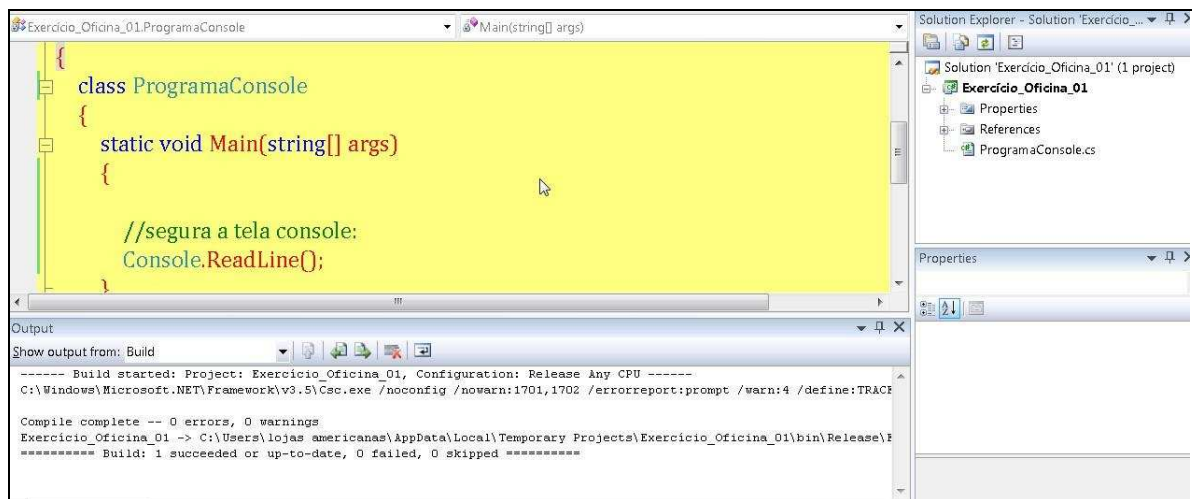


figura 9: a janela output mostrando os principais avisos de compilação.

A linguagem C# precisa da inicialização de seus campos, diferentemente de algumas linguagens como o VBA cuja falta de inicialização acarreta a atribuição de valores default. Vamos dar um exemplo:

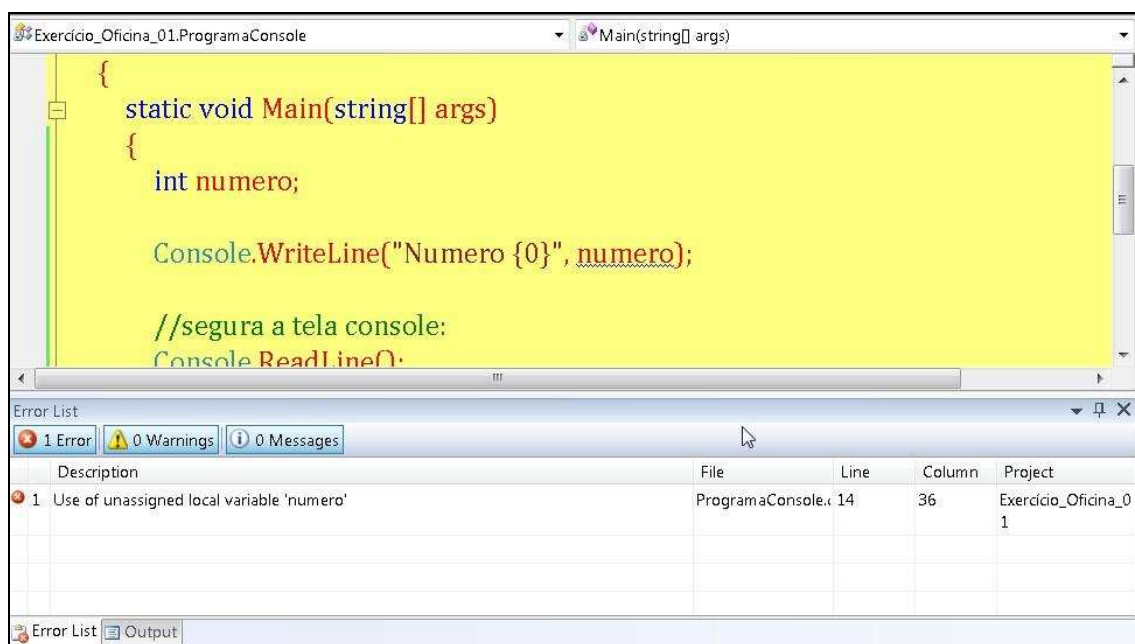


figura 10: falta da inicialização do campo numero.

Acima, nós criamos uma variável (campo), contida na principal classe da aplicação, dentro do método **Main**. Esta variável foi declarada adequadamente, mas não foi inicializada. Ao clicarmos em Build, o output mostra os erros em tempo de compilação. A frase: *Use of unassigned local variable 'numero'*, significa que o compilador acusou erro em **tempo de compilação** devido à falta de atribuição de valor inicial a esta variável. Se atribuirmos algum valor inicial à esta variável, a execução (F5) da aplicação gera a saída da janela console como pode ser vista ao lado.

O projeto foi construído, compilado, o código IL foi criado e o executável foi gerenciado pelo sistema. O resultado do programa é comparado com o seu código fonte, na figura abaixo:

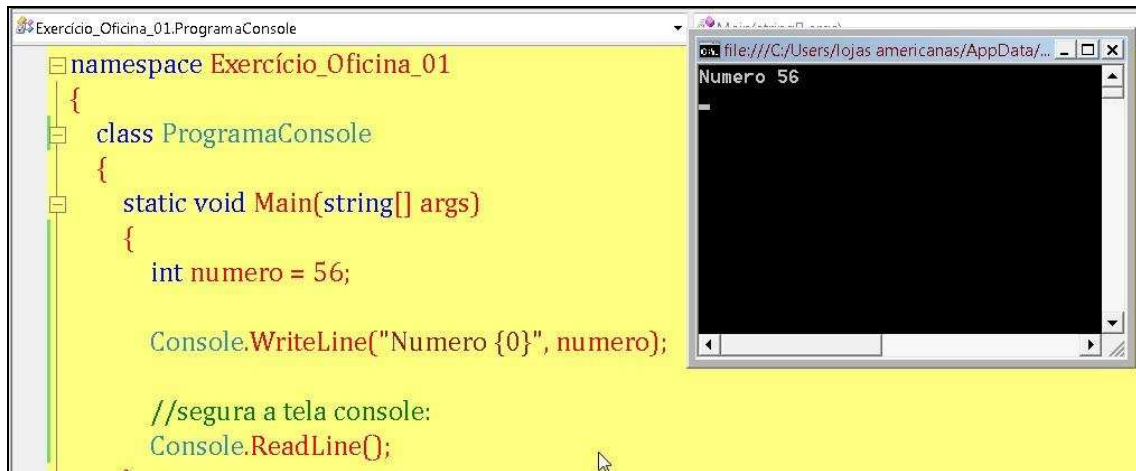


figura 11: a aplicação em execução, juntamente com o seu código-fonte.

Entrada de Dados

A classe **Console** é uma classe da plataforma NET que possui muitos métodos, entre eles os principais correspondem à entrada e saída de dados. Os principais métodos são `.Readline` e `.Writeline`, os quais são exemplificados abaixo:

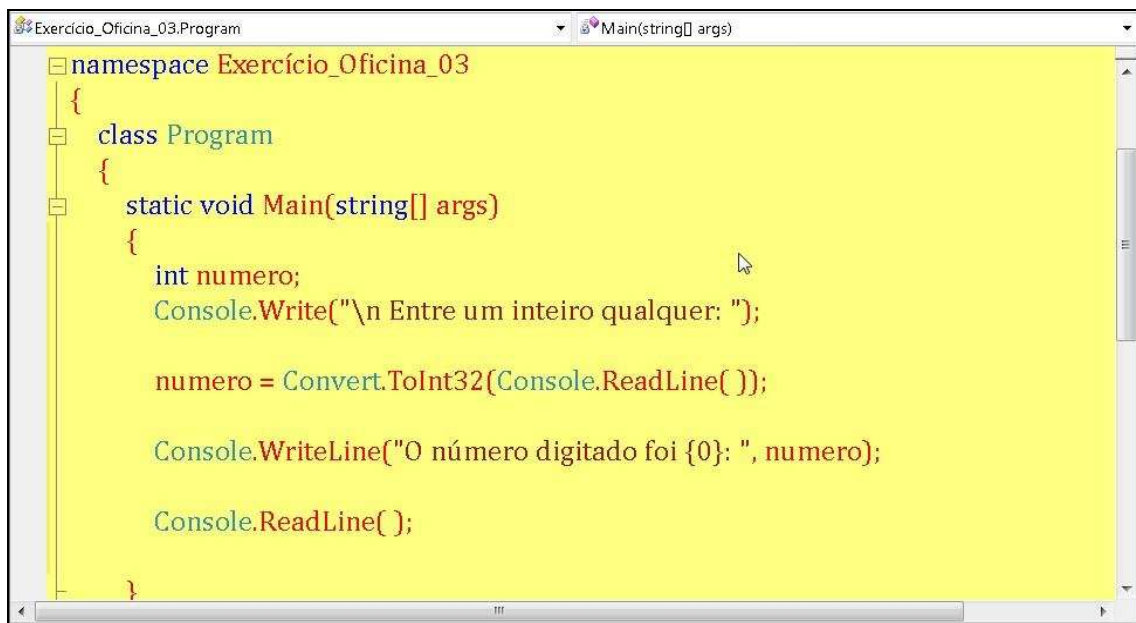


figura 12: entrando dados digitados pelo usuário.

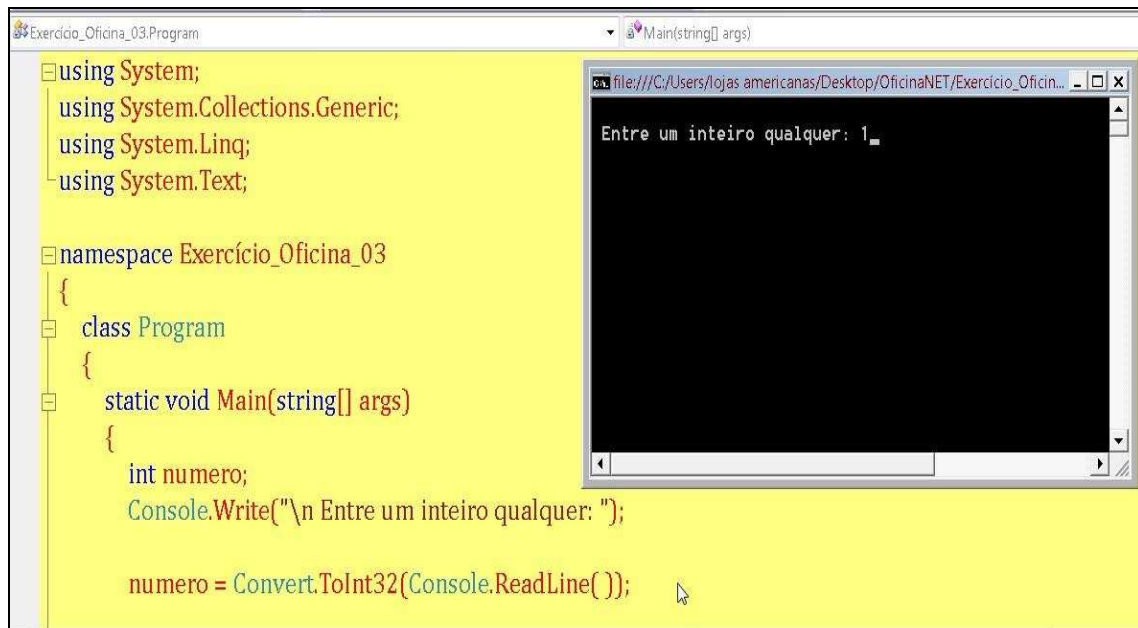


figura 13: rodando o aplicativo de entrada de dados.

Métodos

Um **método** é um bloco nomeado de código executável, que pode ser executado a partir de diferentes partes do programa, e mesmo a partir de outros programas. Quando um método é chamado, ele é executado e então retorna algo ao código que o chamou. Alguns métodos retornam valor à posição a partir da qual eles foram chamados. Métodos correspondem a funções membro em C++.

A mínima sintaxe para a criação de um método é a seguinte:

- **Tipo de retorno** – Estabelece o tipo de valor que o método retorna. Se um método não retorna nada ele é de tipo void.
- **Nome** – Especifica o nome que você escolheu para o método.
- **Lista de parâmetros** – Consiste no mínimo de um conjunto vazio de parênteses, os parâmetros devem estar contidos nestes parênteses.
- **Corpo do método** – Consiste de alguma lógica de código executável.


```

class Program
{
    static void Main(string[] args)
    {
        //a lista de parâmetros é vazia:
        Escreve( );
    }

    private static void Escreve( )
    {
        Console.WriteLine("1");
        Console.WriteLine("2");
    }
}

```

figura 14: Método sem retorno, com modificador de acesso private.

Métodos e Parâmetros

Até agora vimos que métodos são unidades de código que podem ser chamadas a partir de muitos lugares de seu programa, e podem retornar um único valor ao código chamador. Retornar um único valor certamente é valioso, mas o que ocorre se você decide retornar mais de um valor ao código principal? Se você deseja retornar múltiplos valores, você deve usar parâmetros. Pode ocorrer de você desejar passar dados a um método quando ele inicia execução. Parâmetros são variáveis especiais utilizadas para realizar ambas as tarefas. Os parâmetros passados entre os métodos são divididos em formais e reais. Vamos explicar a diferença entre eles:

Parâmetros Formais

São variáveis locais que são declaradas na lista de parâmetros do método, mais que no corpo do método.

```

//Encapsulamento:(você pode adicionar outras
//funcionalidades ao método privado abaixo:
private static void EscreveNúmeros(int x, float y)
{
    Console.WriteLine(" Num {0}, {1}", x, y);
}

```

- Por que parâmetros formais são variáveis, eles possuem tipo de dado e um nome, e podem ser escritos e lidos a partir.
- A lista de parâmetros pode conter qualquer número de itens, separados por vírgulas.

- Podem ser usados para definir outras variáveis locais.

```
//A assinatura do método é diferente:
private static void EscreveNúmeros(int x, float y, float soma)
{
    Console.WriteLine(" Num {0}, {1}, {2}", x, y, soma);
}
```

figuras 15 e 16: parâmetros formais e reais.

Parâmetros Reais

Quando seu código chama um método, os valores dos parâmetros formais podem ser inicializados antes que o código dos métodos comece a execução.

Os parâmetros reais são inseridos na lista de parâmetros da invocação do método. Devem ser observadas as seguintes regras:

- O número de parâmetros reais deve concordar com o número de parâmetros formais (só há uma exceção a esta regra).
- Cada parâmetro real deve ser do mesmo tipo do correspondente formal.

O programa **Parâmetros_Reais** apresenta o exemplo acima. Inicie uma aplicação Console e renomeie Program.cs como CalculaSomaMedia.cs. Crie uma classe denominada MétodosAuxiliares.cs e escreva dois métodos públicos com valores de retorno. Uma retorna tipo float e a outra retorna tipo int. Veja a codificação abaixo e a IDE:

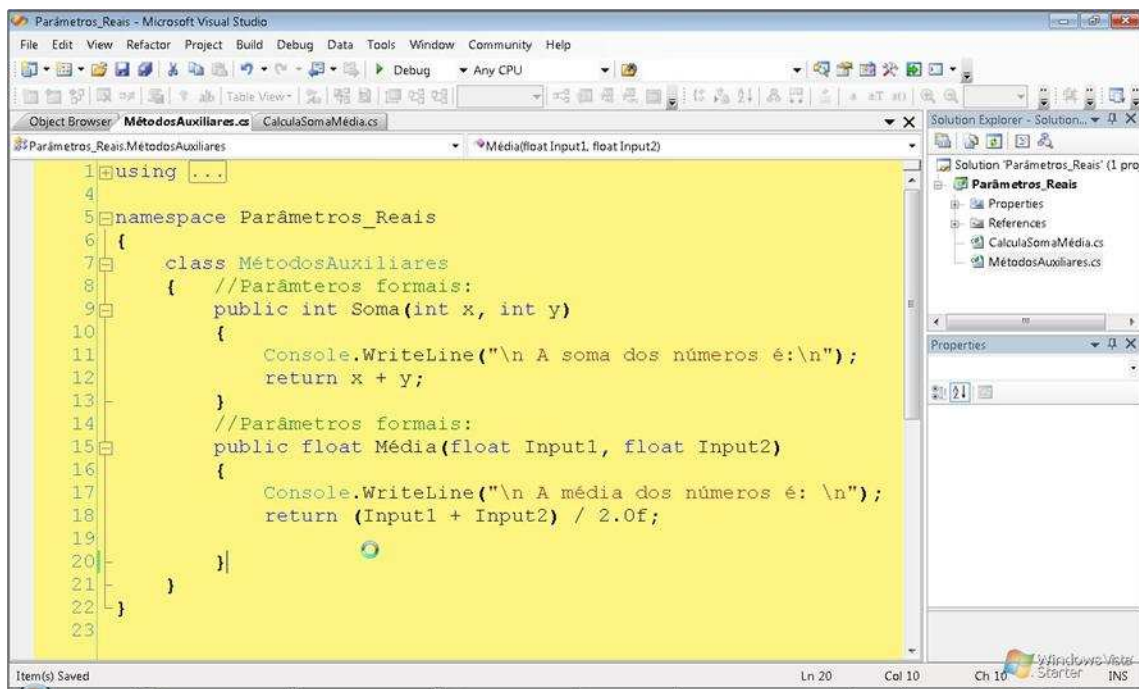


figura 17: dois métodos da classe MétodosAuxiliares.cs.

Em seguida, a classe que contém Main possui o seguinte código:

```
1 using ...
4
5 namespace Parâmetros_Reais
6 {
7     class CalculaSomaMédia
8     {
9         static void Main(string[] args)
10        {
11
12            MétodosAuxiliares m = new MétodosAuxiliares();
13            int algum = 6;
14
15            //parâmetros reais (m é objeto e 5 e algum são parâmetros reais:
16
17            Console.WriteLine("Soma {0} e {1} = {2}", 5, algum, m.Soma(5, algum));
18            Console.WriteLine("M de {0} e {1} = {2}", 5, algum, m.Média(5, algum));
19
20            Console.ReadLine();
21        }
22    }
23 }
24
```

figura 18: o método Main chama os métodos definidos na classe MétodosAuxiliares.cs.

Tipos de dados primitivos e operadores

A linguagem C# contém uma série de operadores unários e binários, necessários nesta e na verdade em todas as linguagens de programação. Os operadores realizam múltiplas tarefas: reverterem o conteúdo lógico de uma variável booleana de true para false, adiciona dois números, concatena duas strings, assim por diante. Vamos apresentar algumas definições muito recorrentes:

Identificadores: são nomes que você usa para identificar os elementos em seu programa. Os identificadores devem começar por letras minúsculas ou maiúsculas, nunca por números, ex: NomeJogador, ContagemScore, assim por diante.

C# é uma linguagem case sensitive, de modo que: contagemJogador e ContagemJogador são identificadores diferentes. VB não é case sensitive.

A linguagem C# admite palavras reservadas (keywords) as quais não podem ser atribuídas a identificadores.

Palavras reservadas (keywords)

A linguagem C# usa 77 palavras-chave consideradas reservadas (ou keywords). Elas não podem de nenhuma forma serem usadas em outro contexto, como por exemplo se você deseja usar uma palavra reservada como nome de uma variável. Em outras palavras, o uso destas keywords é de restrição funcional e depende do contexto lógico para sua utilidade. Aqui estão elas:

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

tabela 2: palavras reservadas de C#.

NOTA: Na janela de edição do **Visual Studio 2008**, as palavras-chave são coloridas em azul quando você as digita.

A linguagem C# possui um conjunto de dados primitivos. Quando você declara uma variável que armazena um tipo primitivo, deve escrever: tipo_dado nome_variável; exemplo: Int númeroEmpregados;

A seguinte tabela apresenta o conjunto de dados primitivos da linguagem C# e o intervalo de capacidade de cada tipo de dado. A penúltima coluna mostra o quanto cada dado deve alocar na memória.

Tipo de dado	Descrição	Tamanho(bits)	Intervalo(range)
Int	Números inteiros	32	$[-2^{31}, 2^{31} - 1]$
Long	Números inteiros	64	$[-2^{63}, 2^{63} - 1]$
Float	Ponto flutuante	32	$[\pm 1.5 \times 10^{-45}, \pm 3.4 \times 10^{38}]$
Double	Precisão dupla	64	$[\pm 5.0 \times 10^{-324}, \pm 1.7 \times 10^{308}]$

Decimal	Valores monetários	128	28 dígitos significativos
String	Seqüência de caracteres	16/caractere	Não aplicável
Char	Caracter único	16	0 a $2^{16} - 1$
Bool	Lógico	8	True ou False

tabela 3: tipos de dados primitivos em C# e sua alocação em memória.

Operadores Binários e Unários

Uma linguagem de programação não é de utilidade se ela não é capaz de realizar a composição de tipos de dados primitivos (ou definidos pelo usuário) em novos tipos de dados. Para isto, as linguagens possuem operadores que permitem cálculos (somar, multiplicar e dividir) matemáticos e comparações, e mesmo operações de ordem lógica. A linguagem é finalmente completada com o uso dos controles de fluxo.

Os **controles de fluxo** são blocos lógicos que realizam as operações de decisão, desvio condicional e estruturas iterativas e repetitivas. Não devem ser confundidos com os operadores binários e unários, os quais apresentamos na tabela abaixo:

Categoria	Operadores	Descrição	Associatividade
Primário	++	Pós incremento	Esquerda
	--	Pós decremento	
Unário	!	Lógico não	Esquerda
	+	Adição	
	-	Subtração	
	++	Pré incremento	
	--	Pré decremento	
Multiplicativo	*	Multiplicação	Esquerda
	/	Divisão	
	%	Resto de divisão	
Aditivos	+	Adição	Esquerda
	-	Subtração	
Relacionais	<	Menor	Esquerda

	<=	Menor ou igual	
	>	Maior	
	>=	Maior ou igual	
Igualdade	==	Igual	Esquerda
	!=	Diferente	
Condicionais	&&	E	Esquerda
		Ou	
Atribuição	=	Atribuição	Direita

tabela 4: Os tipos de operadores da linguagem C#.

A tabela acima pode ser melhor compreendida através da construção de alguns exemplos. Devemos fazer algumas observações: os operadores = e == em C# são muito diferentes em significado, embora à primeira vista possam indicar quase o mesmo tipo de semântica.

Vamos começar com o **operador de atribuição** (=). Este operador é usado para atribuir valor a uma variável (seja valor fixo ou definindo uma equação). A sua associatividade é direita, isto é, o valor a ser atribuído está à direita da variável. Vamos a um exemplo:

Int32 numero = 45;

Neste caso, declaramos uma variável do tipo Int32, a qual denominamos numero. A esta variável nós atribuímos o valor 45. Esta atribuição é da direita para a esquerda, você deve entender que a variável numero recebe o valor 45.

O operador == indica uma igualdade de fato. Veja o exemplo abaixo:

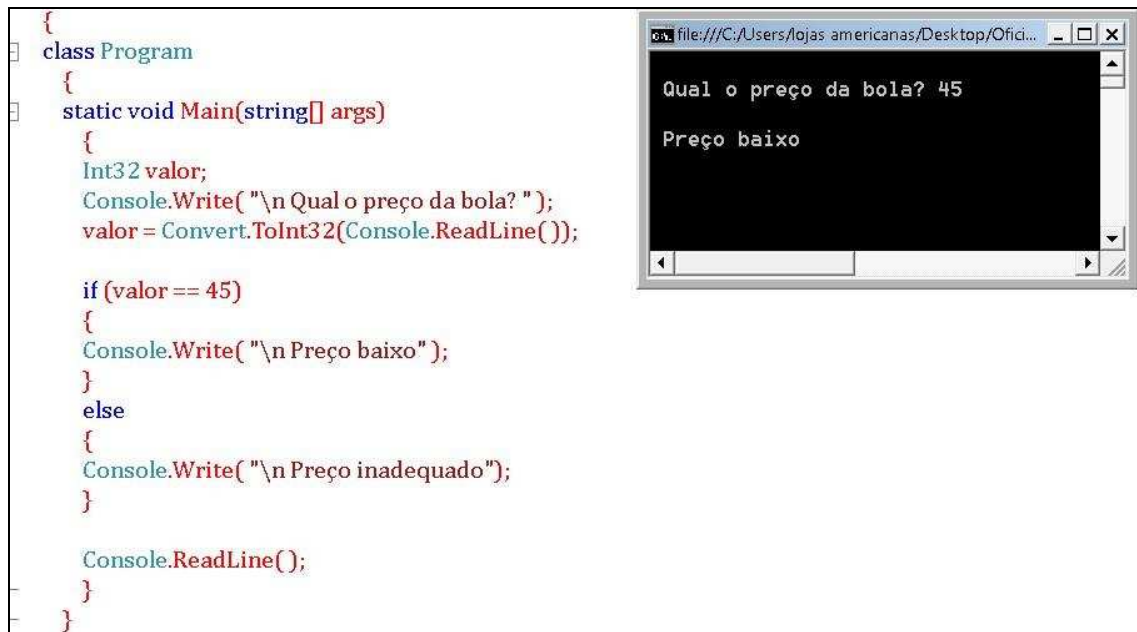


figura 19: código fonte e execução do mes mo.

Explicação do aplicativo

A pequena aplicação console acima pede para o usuário adivinhar o preço de uma bola. Se (if) ele digita 45, aparece a mensagem na tela: Preço baixo, em caso contrário a tela mostra outra frase: Preço inadequado. Note que, dentro do controle de fluxo if(...) foi usado o operador == que indica a igualdade do conteúdo do valor da variável valor com o número 45. Sendo esta igualdade verdadeira, o valor lógico da expressão valor == 45 passa a ser true (verdadeiro) e este controle de fluxo salta à expressão Preço baixo, e não executa Preço inadequado (o qual é chamado por else (senão)). Capturamos em tela a execução console para mostrar o seu funcionamento. É importante perceber que não podemos escrever valor = 45 no controle de fluxo if. Tal construção não envolve uma igualdade, ela atribui 45 à variável valor e se anteriormente fosse outro valor, seria trocado devido à esta construção.

Controles de Fluxo

As linguagens de programação modernas possuem estruturas de **controle de fluxo**, as quais são usadas para redirecionar o fluxo de dados e passagens de parâmetro ao longo do programa. Mais de 90% das linguagens de programação modernas são **estruturadas** pois possuem controles de fluxo contendo for., select.. case, if, do...while, while, e estruturas de tratamento de erro como try..catch.

Até algumas versões mais recentes do SQL são hoje em dia **estruturadas** pois agregam **if then else** e **select case** entre outras estruturas. Uma linguagem é considerada **estruturada** quando contém estes elementos para o controle do fluxo dos dados, apenas isto, não dependendo de uma vinculação com relação à posição inicial de variáveis, tais

opções são mais restrições de compilador e especificações muito particulares da linguagem.

O importante aqui é que as diversas linguagens de programação são classificadas em termos de critérios mais complexos, se são **orientadas a objeto, orientadas a aspecto, dirigidas a evento, funcionais, lógicas**, etc.

Mas **todas estas linguagens** são estruturadas, pois nenhuma linguagem moderna e realmente útil deixa de ser estruturada, e desta forma este não constitui critério de classificação importante. Algumas poucas linguagens não são estruturadas, mas não de importância para a maioria das aplicações mais modernas.

Algumas linguagens não são estruturadas como o XML mas XML é uma **linguagem de marcação de dados** e o seu tipo de organização de estruturas é diferente do estabelecido pelas **linguagens de programação**.

Nesta apostila não vamos explicar os demais critérios que classificam as linguagens em imperativas, declarativas, ortogonais, etc, e remetemos o leitor ao excelente livro de Robert Sebesta: *Conceitos de Linguagens de Programação*, onde tais conceitos são plenamente explicados e apresentados.

As estruturas de fluxo cumprem algumas tarefas básicas: repetição de uma tarefa um número especificado de vezes, repetição de tarefa se uma dada condição é verdadeira, desvio condicional, desvio lógico e seleção de casos. Tais controles usam evidentemente palavras reservadas da linguagem. A primeira estrutura de controle que apresentamos agora é **if else**.

1) If Else

Esta estrutura é usada para desviar condicionalmente entre duas tarefas distintas, de acordo com uma condição ser ou não verdadeira. Veja o exemplo console abaixo:

```
Int32 numero = 12;
Int32 palpite;

Console.Write( "\n Entre um número entre 10 e 20 escolhido pelo computador: ");
palpite = Convert.ToInt32(Console.ReadLine());

if (palpite == numero )
{
    //Tarefa1:
    Console.Write( "\n Acertou! Dez pontos para você ");
}
else
{
    //Tarefa2:
    Console.Write( "\n Você errou, tente em outra ocasião ");
}
```

Comentários

O programa armazena um número inteiro e o inicializa em 12. Em seguida ele solicita ao usuário digitar um número pelo teclado. Caso ele digite 12 e tecle ENTER em seguida, o console apresenta a frase em tela: Acertou! Dez pontos para você. Se o usuário digita qualquer outro número diferente de 12, a condição é falsa e é necessariamente desviada para o **else**, onde o console escreve: Você errou, tente em outra ocasião. Note que em nenhum momento são executadas as duas tarefas ao mesmo tempo, através deste programa. Isto é porque denominamos **desvio condicional**.

Caso a condição **palpite == numero** seja verdadeira (true), é executada a tarefa //Tarefa1: se a condição **palpite == numero** é falsa, a estrutura desvia para o else e executa //Tarefa2: (O que está escrito em verde na forma: //(comentário) não é lido pelo compilador, é um comentário, entraremos em detalhes sobre documentação e comentários mais adiante).

O laço de repetição for(início; término; step ou passo)

2) for()

Esta estrutura é usada para realizarmos o mesmo número de tarefas um número especificado de vezes. Vamos supor que o usuário deseja repetir a mesma tarefa dez vezes, ele escreve a estrutura como:

```
for(int J=1; J<=10; J++)  
{  
    Tarefa(...);  
}
```

J++ significa: J = J+1, ou seja, a cada iteração o parâmetro inteiro J é somado de uma unidade. Quando J = 10 o laço termina a sua execução. Aparentemente, parece que a Tarefa(...) é executada da mesma forma.

Isto não é verdade, se a tarefa possuir parâmetros que serão atualizados a cada iteração, a Tarefa(...) é repetida mas pode gerar saídas diferentes em cada iteração.

O passo não precisa ser necessariamente J++, podemos escrever algo do tipo:

```
for(int J = 1200; J > 10; J=J-4)
```

O contador começa em 1200 e vai decrementando de 4 em 4 unidades até alcançarmos 10, quando o laço termina.

3) seleção múltipla: switch() {case break; }

Uma alternativa muito elegante à estrutura if else ocorre quando devemos escolher entre diversas alternativas, como se fossem casos. Esta estrutura é mais elegante do que se você aninhar if else dentro de um if else. Note que os casos devem ter uma hierarquia equivalente, você pode aninhar if else dentro de algum case, ou outra estrutura de seleção. Veja o exemplo de código abaixo, usando a estrutura switch:

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

namespace seleção_múltipla
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\n Escolha seção da Loja
(menu) : ");

            Console.WriteLine("1: perfumaria");
            Console.WriteLine("2: roupas");
            Console.WriteLine("3: lingerie");
            Console.WriteLine("4: chocolates");

            Console.Write("\n Seção escolhida: ");
            Int32 seção =
Convert.ToInt32(Console.ReadLine());

            switch (seção)
            {
                case 1:
                    Console.WriteLine("\n Você está na seção de
perfumaria.");
                                //mais código
                    break;

                case 2:
                    Console.WriteLine("\n Você está na seção de
roupas.");
                                //mais código
                    break;
```

```

        case 3:
            Console.WriteLine("\n Você está na
seção lingerie.");
            //mais código
            break;

        case 4:
            Console.WriteLine("\n Você está na
seção chocolates.");
            //mais código
            break;

        default:
            Console.WriteLine("\n Não há esta
seção na Loja!");
            break;

    }

    Console.WriteLine("\n FIM do Programa");
    Console.ReadLine();

}

}

}

```

O usuário escolhe uma das seções, digitando um número inteiro entre as opções acima, se o número não corresponder a uma das opções o console apresenta Não há esta seção na Loja!

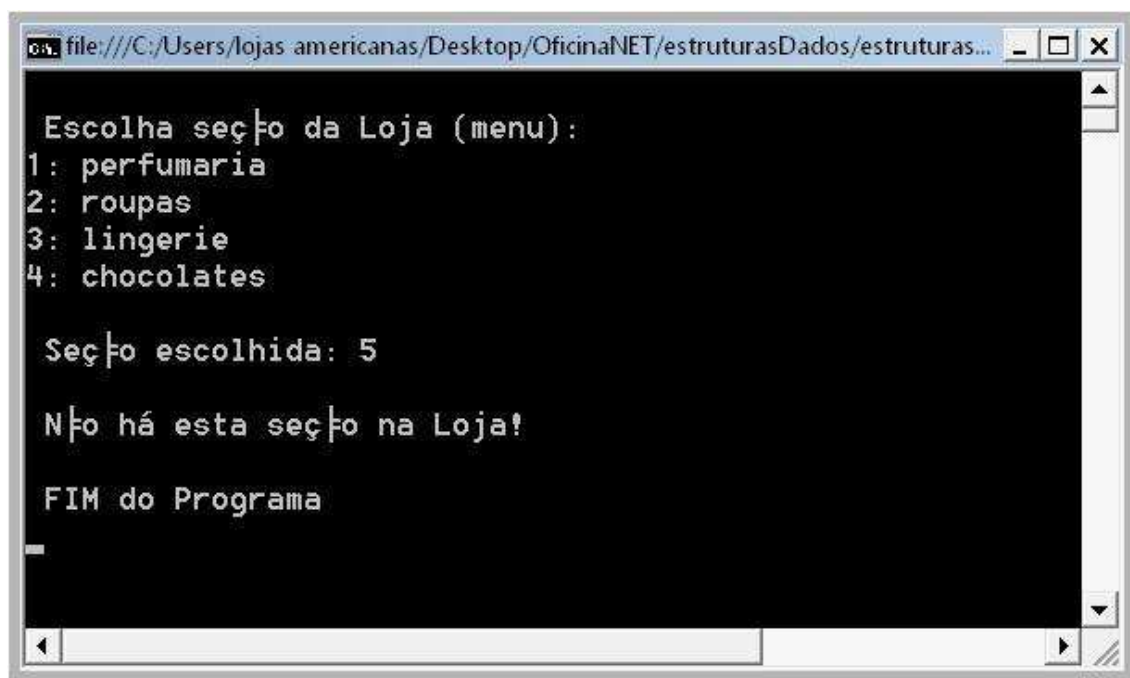


figura 20: menu de escolhas indexado por inteiro.

A estrutura **switch** aceita os tipos primitivos e string, caso você deseje que o usuário escolha uma opção como texto. Note que esta linguagem não permite que você escolha como um case um intervalo entre os números inteiros. A linguagem **VBA** usada no **Excel** e nos demais aplicativos **Office** permite este tipo de construção. Todo case desta estrutura exige break; e a declaração default ...break; não precisa ser colocada por último necessariamente, podendo ser colocada em qualquer posição dentro do **switch**.

4) Faça enquanto: **do{ }while();**

Esta estrutura realiza o código entre chaves enquanto uma condição continua sendo verdadeira. Em outras palavras, no parênteses que acompanha while() há uma condição que deve ser avaliada como verdadeira ou falsa, o laço termina quando a condição for falsa. A montagem da estrutura **do while** deve ser executada com cuidado pois ela pode ser fonte de laços infinitos e neste caso a execução nunca termina.

Uma aplicação tipo console que almeja ser razoavelmente útil deve compreender um grande do while envolvendo todo o código fonte principal. Isto é muito simples de entender: aplicações tipo windows permitem que você retorne a alguma caixa de texto e apague o seu conteúdo, você clica novamente nos botões de comando e reinicia as atividades, com a aplicação console isto não ocorre, para você retornar à entrada de dados inicial você deve usar do while(); vejamos um exemplo console abaixo:

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

```

namespace doWhile
{
    class Program
    {
        static void Main(string[] args)
        {
            String retorno;

do
        {
            Console.WriteLine("\n Digite numero inteiro: ");
            Int32 numero = Convert.ToInt32(Console.ReadLine());

            if (numero % 2 == 0)
            {
                Console.WriteLine("Numero {0} e par", numero);
            }
            else
            {
                Console.WriteLine("Numero {0} e ímpar", numero);
            }

            Console.WriteLine("\n Deseja repetir
operação?(s/n)");
            retorno = Console.ReadLine();

        } while (retorno == "s" || retorno == "S");

            Console.WriteLine("\n FIM do Programa");

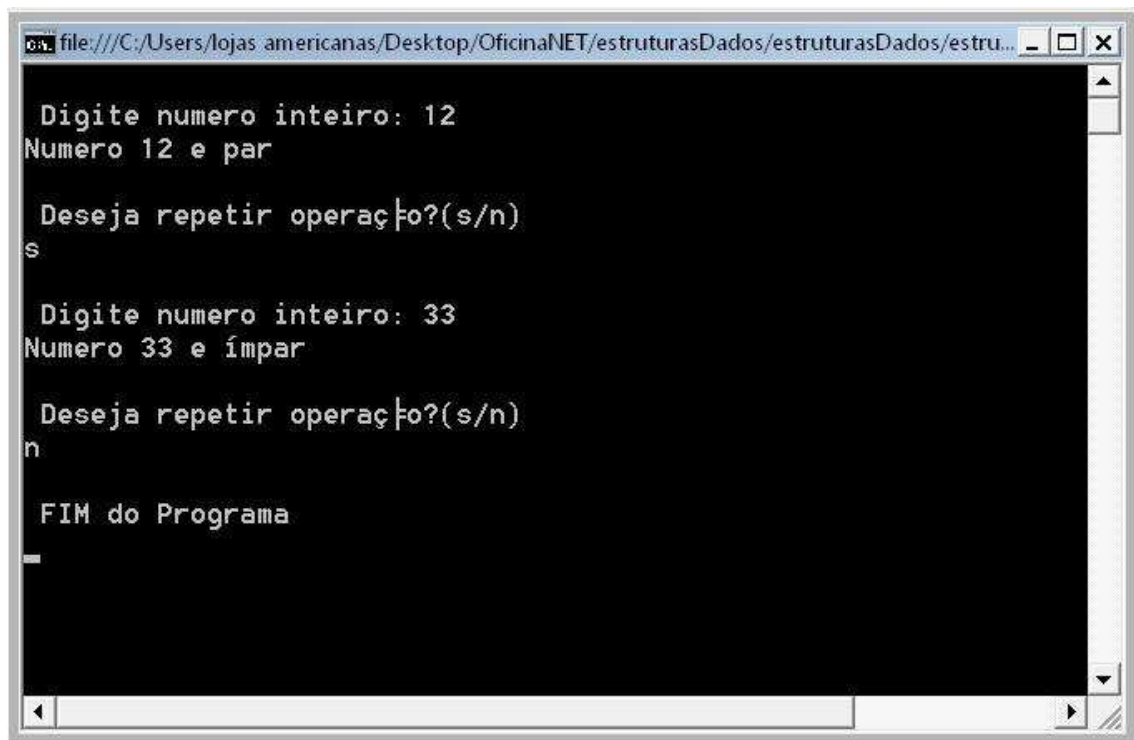
            Console.ReadLine();
        }
    }
}

```

Neste exemplo **Console**, o aplicativo solicita a digitação de um número inteiro, quando o usuário entra um número inteiro qualquer, o programa calcula se o mesmo é ímpar ou par (% significa resto inteiro da divisão), o programa em seguida pergunta se você deseja repetir a operação com outro número. Se você digita s ou S o programa retorna à linha na qual o mesmo pede a entrada de número inteiro. O programa realiza a determinação de pares e ímpares enquanto você responde s ou S (condição é verdadeira no while).

Quando você digitar qualquer outra coisa, esta condição será falsa e o aplicativo sai do laço do while e termina a sua execução.

Veja a sua execução:



```
file:///C:/Users/lojas_americanas/Desktop/OficinaNET/estruturasDados/estruturasDados/estru...
Digite numero inteiro: 12
Numero 12 e par
Deseja repetir operaç|o?(s/n)
s
Digite numero inteiro: 33
Numero 33 e ímpar
Deseja repetir operaç|o?(s/n)
n
FIM do Programa
```

figura 21: execução do aplicativo seleção de paridade.

Juntando as peças: faça um acumulador de apostas.

Agora, vamos a um conceito importante que é o conceito de aninhamento ou **encadeamento** de estruturas. Todas as linguagens estruturadas permitem o encadeamento de estruturas umas dentro das outras (daí o nome de programação estruturada), logo, podemos encadear um if else dentro de um do while por exemplo.

Para sentirmos o gostinho deste tipo de estrutura, pense no seguinte aplicativo **Console**: você entra um número inteiro entre 0 a 10, o computador gera um número aleatório (o qual você não tem acesso) e compara o seu palpite com o número gerado. Caso o tento seja bem sucedido você marca um ponto, caso contrário informa que o palpite foi errado. O programa deve permitir que você repita a tentativa com outro número inteiro dentro do intervalo permitido (0 a 10).

Quando você terminar o número de apostas, que pode ser arbitrário, o programa deve gerar um relatório do índice porcentual de acertos. Veja abaixo uma solução possível para este problema:

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



```

namespace Palpite
{
    class Program
    {
        static void Main(string[] args)
        {

            int tento = 0;
            int pontos = 0;
            string retorno;

do
{

    Random r = new Random();
    int aleatorio = r.Next() % 10;


    Console.WriteLine("\n Adivinhe o número escolhido pelo
computador?(0 a 10): ");

    int palpite = Convert.ToInt32(Console.ReadLine());


    if (palpite == aleatorio && (palpite <= 10 && palpite
    >= 0))
    {
        Console.WriteLine("\n Número gerado: {0}", aleatorio);
        pontos += 1;
        Console.WriteLine("\n Você marcou {0} ponto. ", pontos);

    }

    else
    {
        Console.WriteLine("\n Número gerado: {0}", aleatorio);
        Console.WriteLine("\n Palpite errado desta vez");
    }

    tento += 1;

    Console.WriteLine("\n Deseja repetir aposta?(s/n)");
    retorno = Console.ReadLine();
    Console.Clear();

} while (retorno == "s" || retorno == "S");

```

```

        Console.Write("\n Pontos: {0}", pontos);
        Console.Write("\n Tentativas: {0}", tento);

        Console.Write("\n Índice de acertos = {0} por
cento", ((double)pontos / (double) tento)*100);

        Console.WriteLine("\n FIM do Programa.");
        Console.ReadLine();

    }
}
}

```

Comentários

Para efeito de simplificação, não fizemos o tratamento de erros neste programa, que será adiante explicado e também esta não é a única ou melhor solução (a informação de um número fora do intervalo deveria ser melhor arquitetada), mas é claro que o objetivo da apresentação do mesmo é para fins apenas didáticos. Vamos explicar os seus pontos principais:

- 1) Começamos com a inicialização de dois inteiros fora do do ...while e inicializados em zero. Estes números devem estar fora do **do ...while**, pois caso contrário a saída (output) não terá acesso aos mesmos.
- 2) Para gerarmos um número aleatório precisamos de uma semente. Esta semente é providenciada pela classe Random. Você inicializa um objeto Random, do tipo inteiro. Para a geração do número, você precisa declarar uma segunda variável, chamada aleatório, a qual pega a semente r e com o método Next() calcula o aleatório e armazena na variável. Para que este número caia no intervalo de 0 a 10, usamos %10 em seguida.
- 3) O programa aguarda a entrada de um inteiro e em seguida compara com o aleatório que foi determinado acima. No if...else, o programa determina se houve acerto e o acumulador pontos é atualizado em +1. O acumulador tento sempre deve ser atualizado, a cada vez que o loop do... while é executado.
- 4) O loop do ... while (o qual contém o if que está encadeado) é retornado apenas com o consentimento positivo do usuário. Se ele digitar qualquer outra coisa, o programa encerra este laço e sai do mesmo.
- 5) Finalmente o desempenho estatístico é apresentado após o término deste laço.

Desafio PUTNAM

Note que um **gerador de números aleatórios** é um componente essencial de uma boa biblioteca de uma linguagem de programação, mas não existe geração perfeita de números aleatórios. Isto é matematicamente demonstrável.

Então, arregace as mangas, consulte os seus livros de estatística, e inspirado pela construção acima construa um aplicativo **Console** que faça a análise de quão perfeitamente aleatório é o gerador Random de C#.

Em outras palavras descubra se e quando há uma tendência de números dentro de um intervalo especificado, números que começam a aparecer mais do que uma amostra aleatória perfeita.

Dicas: não use escolha manual como foi feita acima, aproveite os próprios recursos computacionais dentro de um laço **for** bastante extenso do tipo:

```
for(int J = 0; J<= 1000000; J++)  
{  
    ...código aqui.  
}
```

Elabore um critério de desempenho para cada número de uma faixa escolhida e em seguida apresente um relatório de tendências, do tipo:

- 1) O número 1 apareceu x% vezes dentro do intervalo 1000000.
- 2) O número 2 apareceu y% vezes dentro do mesmo intervalo e assim por diante.
- 3) ...

É razoável que o intervalo acima de 1000000 seja móvel, o usuário escolhe o intervalo de números aleatórios e deve especificar o intervalo da máxima amostragem, então no lugar de 1000000 use uma variável que foi lida da entrada de dados, e faça testes com vários intervalos.

Divirta-se pensando neste problema!

5) Criando e gerenciando Classes e Objetos

Após completar esta seção, você terá contato com os seguintes conceitos:

1. Definir uma classe contendo um conjunto relacionado de métodos e itens de dados.
2. Controlar a acessibilidade dos membros usando as palavras chave `private` e `public`.
3. Criar objetos usando a palavra chave `new` para invocar um construtor.
4. Escrever e chamar seus próprios construtores.
5. Criar métodos e dados que podem ser compartilhados por todas as instâncias da mesma classe usando a palavra chave `static`.

Agora, estamos preparados para irmos um passo além das simples construções de fluxos numa programação não orientada a objetos e considerarmos as poderosas construções que envolvem classes, herança e métodos.

O propósito do encapsulamento

O **encapsulamento** é muito importante ao compreendermos classes. Um programa quando cria uma classe não precisa se preocupar com todos os detalhes de uma dada classe. O programa cria uma instância da classe (objeto) e chama métodos e propriedades que irão atuar sobre aquele objeto. Neste sentido, uma classe funciona como uma caixa preta, escondendo detalhes que podem não ser de interesse para um dado objeto. O encapsulamento tem dois propósitos:

1. Combinar métodos e dados dentro de uma classe, em outras palavras, suportar a classificação.
2. Controlar a acessibilidade dos métodos e dados, em outros termos, controlar o uso da classe.

Construindo sua primeira classe

Inicie o **Visual Studio** e crie um projeto Console. Nomeie-o como **IniciandoClasses**.

No **solution explorer**, clique com o botão direito do mouse no nome do projeto para abrir o menu suspenso que se abre à direita (veja a figura da próxima página).

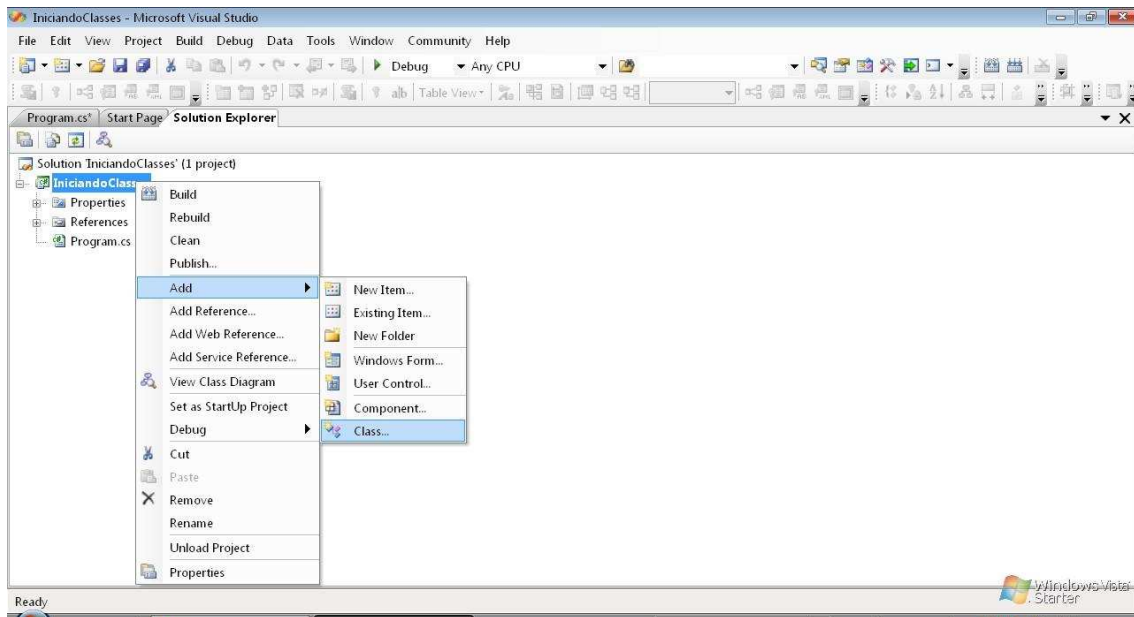


figura 22: criando a sua primeira classe em C# (abra um vinho!)

Adicionando uma classe ao projeto. Após você adicionar a classe, abre-se Add New Item, o qual solicita um nome para a classe, este, deve sempre ter terminação .cs. No nosso exemplo chamamos a classe de **Circle.cs**:

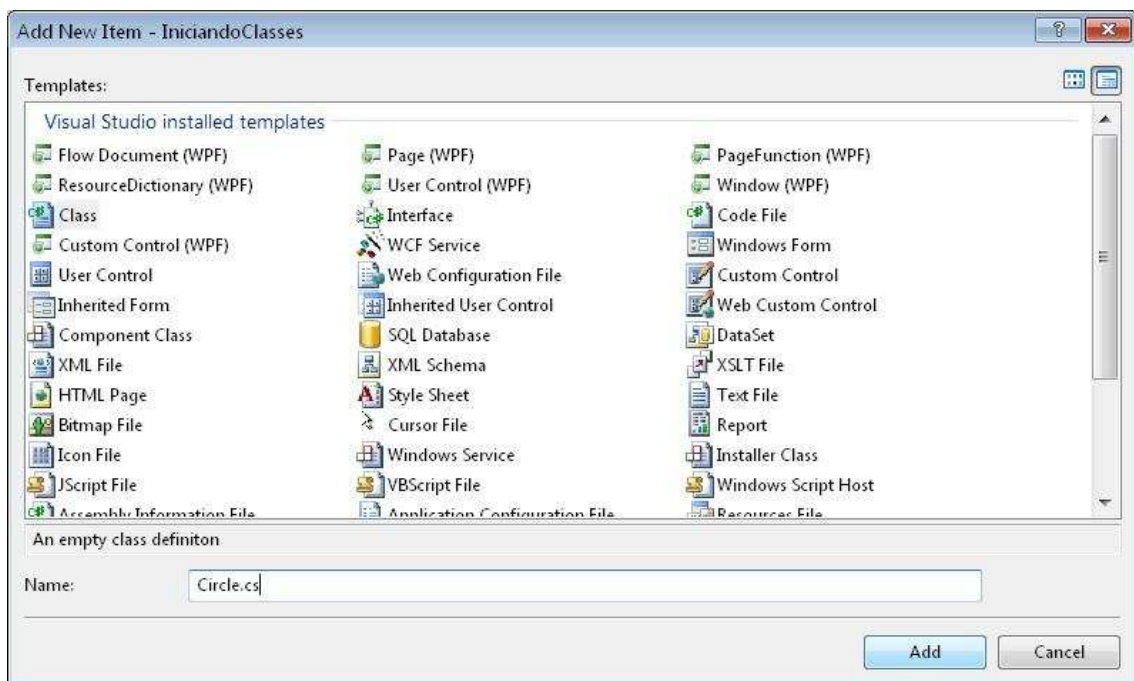


figura 23: definindo a classe Circle.cs.

Finalmente, clique **Add**.

No **solution explorer**, você pode observar que foi acrescentado mais um arquivo para a solução:

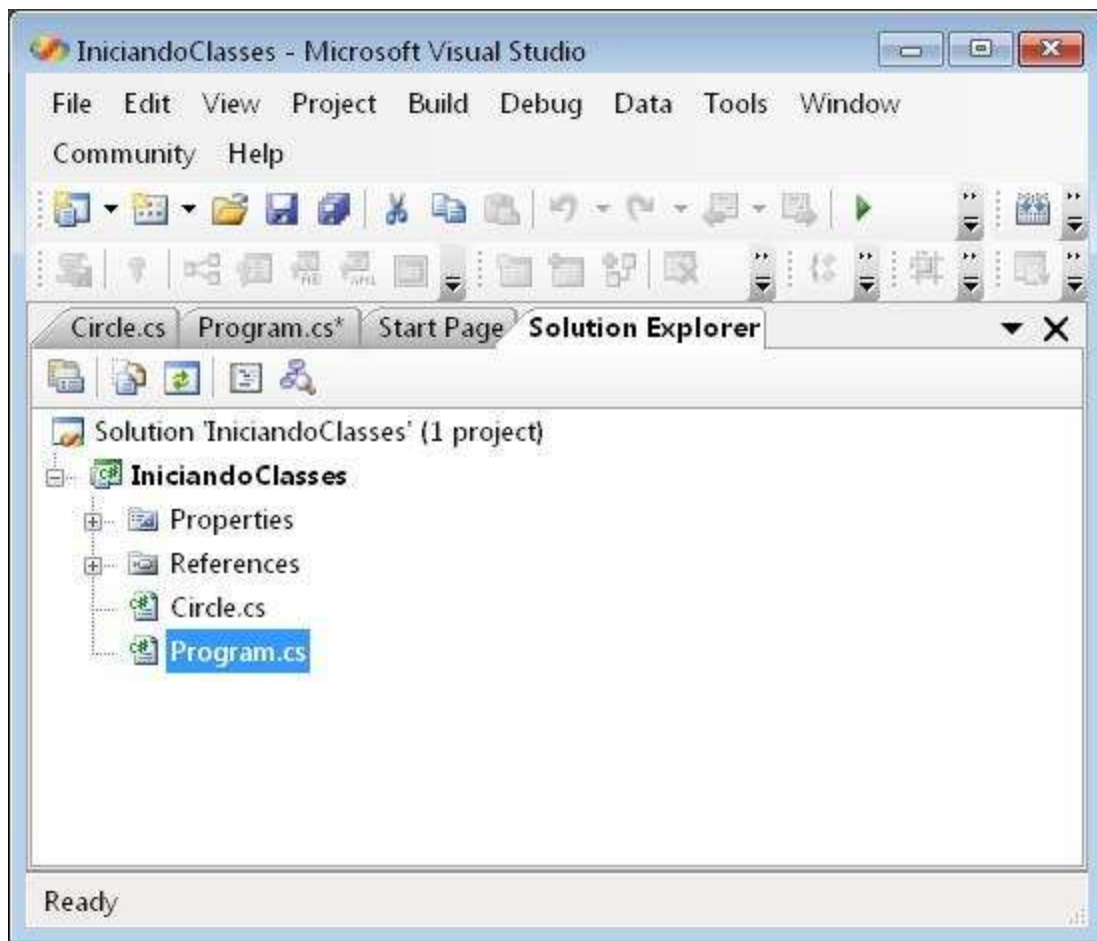
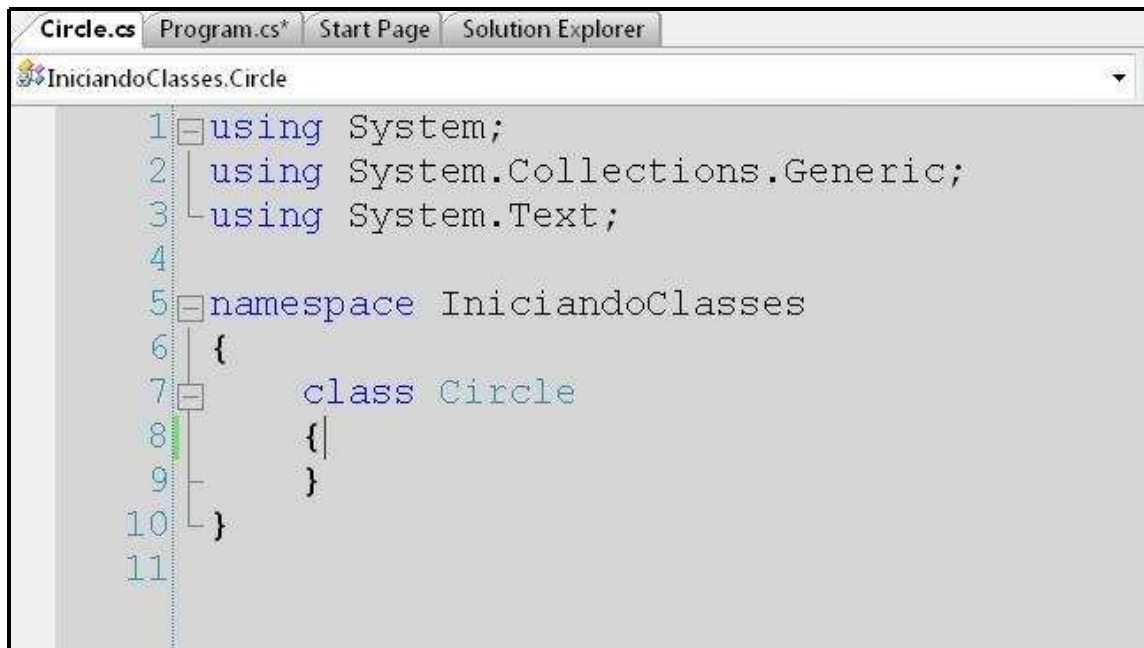


figura 24: localizando Program.cs no seu projeto.

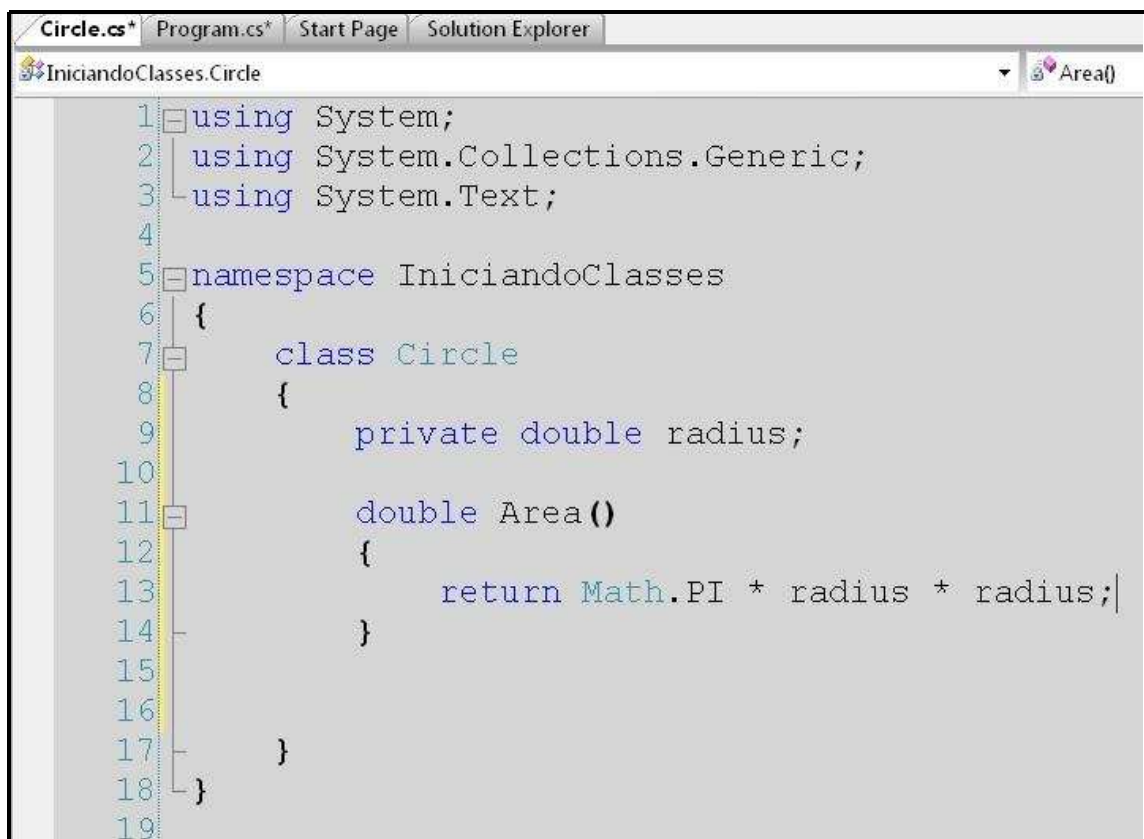
Clique em **Circle.cs** para abrir o arquivo fonte desta classe recém criada (se os números de linhas do código fonte não aparecerem à esquerda, não se preocupe isto é uma questão de configuração do Visual Studio, que explicaremos adiante). O código fonte desta classe é o que está abaixo:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace IniciandoClasses
6 {
7     class Circle
8     {
9     }
10 }
11
```

figura 25: código fonte da classe Circle.cs

O namespace **IniciandoClasses** é simplesmente o nome do projeto que você tinha definido no início. Agora, o código acima não contém qualquer método ou campo. Vamos criar um método que retorna o valor real da área de um círculo de raio qualquer.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace IniciandoClasses
6 {
7     class Circle
8     {
9         private double radius;
10
11         double Area()
12         {
13             return Math.PI * radius * radius;
14         }
15     }
16 }
17
18
19
```

figura 26: iniciando um método com valor de retorno double.

Em C# nós usamos a palavra Class para definir uma classe. Os dados e métodos da classe ocorrem no corpo da classe entre o par de chaves: { }. A classe acima contém um método, denominado Area que retorna um tipo de dado double. O método contém um conjunto de dados: a variável do tipo private com tipo double e denominada radius.

O corpo da classe contém métodos ordinários, tais como Area() e campos, neste caso radius. As variáveis numa classe são denominadas **campos**. Criarmos uma variável ou objeto da classe Circle agimos do modo usual: você cria uma variável especificando que é tipo Circle, e então a inicializa com dados válidos. Para a criação do objeto você deve usar a palavra reservada new pois o objeto não é pré-definido, foi definido pelo usuário.

Note na figura acima que a barra amarela à esquerda significa que a digitação do código acima não foi salvo ou compilado, se você compilar ou salvar a cor torna-se verde. Voltemos à classe **Program.cs**, clicando nela no solution explorer. O seu código-fonte é:

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

namespace IniciandoClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            Circle círculo = new Circle();
            //aloca espaço na memória para uma instância da
            //classe Circle, cuja referência de memória é o
            //identificador círculo.
        }
    }
}
```

Acima, este código-fonte cria a variável **círculo** que é do tipo Circle. Dizemos que este objeto é uma **instância** daquela classe. Você pode atribuir uma instância da classe a outra variável do mesmo tipo:

```
Circle círculo = new Circle();
Circle d;
d = círculo;
```

Controlando a acessibilidade dos membros

A classe que acabamos de criar é entretanto de nenhuma utilidade. O fato é que este objeto é encapsulado numa classe que não fornece acessibilidade a outras regiões do programa. Os dados da classe `Circle` não são de utilidade pois não possuem qualquer visibilidade pelo programa principal. Desta forma, dizemos que estes campos e o método `Area()` são **private**. Entretanto, podemos mudar a acessibilidade destes membros, campos e métodos usando as palavras chave **private** e **public**.

- 1) Um método ou campo é chamado **private** se ele é visível apenas dentro da classe onde está definido. Esta opção é default, mas é boa prática de programação declarar explicitamente quando um dado membro é **private** para evitar confusão.
- 2) Um método ou campo é chamado **public** (público) se ele é acessível a partir de dentro da classe e das outras classes do programa. Voltemos ao código-fonte da classe `Circle` e modifiquemos a sua acessibilidade para **public**.

O código-fonte é:

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

namespace IniciandoClasses
{
    class Circle
    {
        private double radius;
        public double Area()
        {
            return Math.PI * radius * radius;
        }
    }
}
```

Com a palavra-chave **public** o método `Area()` torna-se **público**, isto é, o programa principal pode ter acesso aos campos e os métodos desta classe. Para percebermos que o programa terá acesso ao método `Area()`, escreva a variável `círculo` logo abaixo da definição da instância. Em seguida, quando terminar de escrever `círculo` pressione a tecla do ponto: `.`, quando isto ocorre, aparece um menu (Intellisense) mostrando as opções que podem ser aplicadas para aquela instância (`círculo`).

O **Intellisense** mostra um menu drop-down com todos os métodos disponíveis, sendo que o método `Area()` é aquele definido na classe `Circle`. Vá até este item do Intellisense e aperte ENTER.

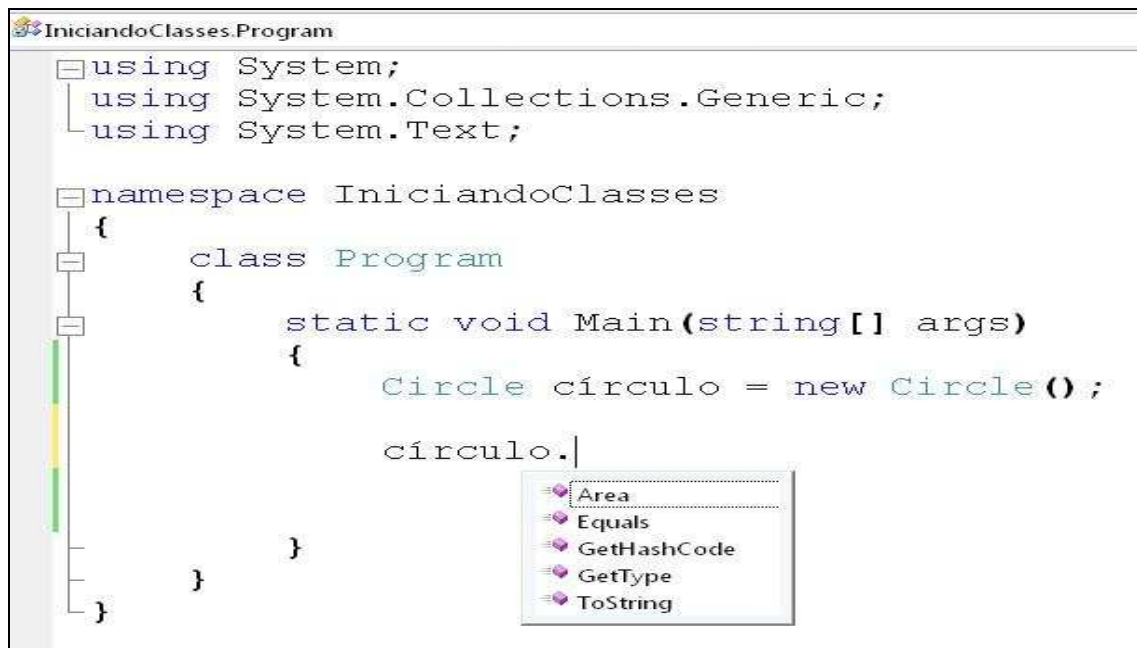


figura 27: o Intellisense em ação: acha o método `Area()` entre os pré-definidos.

Ao clicar em `Area()` veja que aparece um mini-help do lado direito mostrando o tipo de retorno do método, veja a figura abaixo:

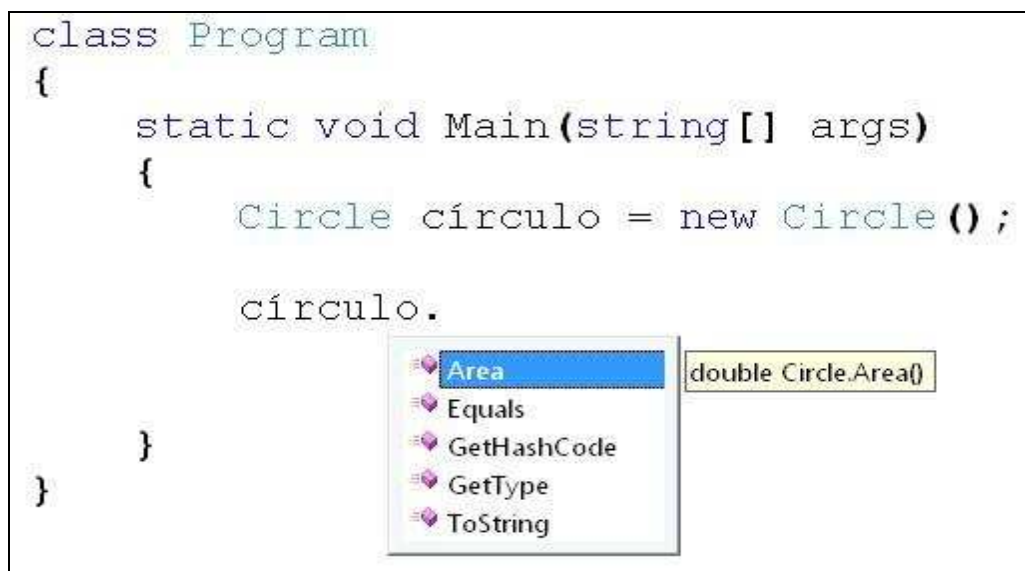


figura 28: selecione `Area()`, aparece um mini-help ao lado direito.

Finalmente, escreva o seguinte código em Main:

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

namespace IniciandoClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            Circle círculo = new Circle();

            círculo.Area();

            Console.WriteLine(círculo.Area().ToString());

            Console.ReadLine();

        }
    }
}
```

Inicializemos o campo radius na classe Circle como abaixo, escolhendo algum valor.

```
public double Area()
{
    radius = 2.0f;
    return Math.PI * radius * radius;
}
```

As linhas do programa:

```
círculo.Area();
Console.WriteLine(círculo.Area().ToString());
```

têm o seguinte significado: a primeira linha simplesmente aplica o método `Area()` ao objeto `círculo`, o qual retorna o valor da área para um raio de valor 2.0. Mas, este método não vai escrever nada em tela. Em seguida, o método `Console.WriteLine()`; irá retornar o valor do método para a tela. `ToString()` é necessário para converter o conteúdo numérico em string, pois tudo em tela deve ser caracteres ou sequência destes. Ao compilarmos e executarmos o programa, obteremos a área de um círculo de raio 2:

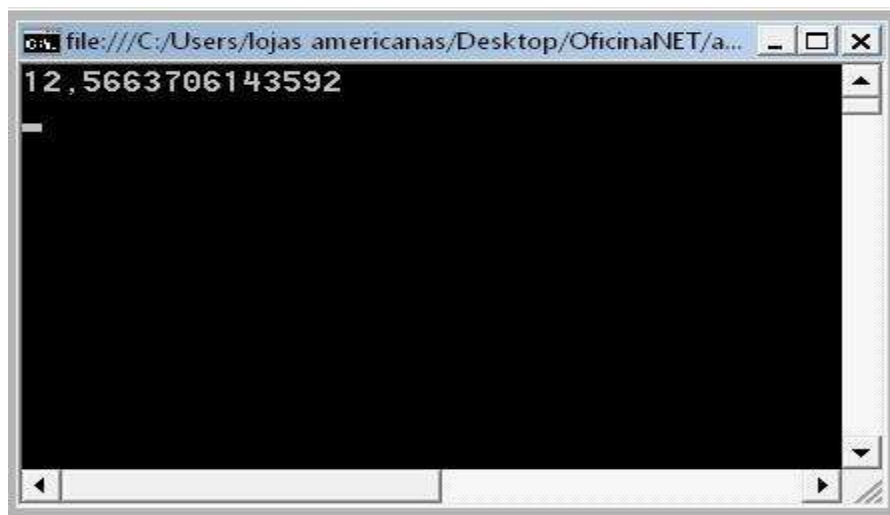


figura 29: resultado da aplicação.

A linha onde está escrito: `Console.ReadLine();` segura a tela, se você não escreve tal linha a tela mostra o resultado e fecha tão rápido que você não captura o resultado. Com esta linha a tela Console pára e espera a próxima ação do usuário.

No código da classe `Circle`, o campo `radius` é definido como `private`, isto significa que este campo é acessível apenas dentro daquela classe. Desta forma, ainda assim, a classe é bastante limitada, digamos que você deseja inicializar um campo através da escolha digitada pelo usuário, como podemos implementar esta tarefa neste programa? A resposta é usar um construtor para a classe, mas antes de entrarmos neste conceito, vamos realizar esta tarefa da seguinte forma: Mude a acessibilidade de `radius` na classe `Circle.cs` para `public` e escreva em `Main` o seguinte código:

```
class Program
{
    static void Main(string[] args)
    {
        Circle círculo = new Circle();

        Console.WriteLine("Qual é o raio do círculo  
escolhido?");
        círculo.radius =
        Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("A área do círculo de raio {0}  
é {1}", círculo.radius, círculo.Area().ToString());

        Console.ReadLine();
    }
}
```

Este código tem 4 linhas com o seguinte significado: 1) A primeira linha, como vimos, cria o objeto círculo. 2) A segunda linha solicita do usuário que ele digite um número (valor) para o raio de um círculo. 3) A terceira linha aguarda a digitação deste valor, pelo usuário, quando ele deve entrar um número qualquer. `Console.ReadLine()` é este comando, mas uma vez que o usuário entra número, o console o percebe como caracteres e não como um número em si, desta forma, `Convert.ToDouble()` se encarrega de converter para número o que foi escrito.

4) Finalmente, após termos entrado com este dado, o método `Console.WriteLine()` escreve o valor do raio e retorna a área que usa este dado. Veja o exemplo em execução:

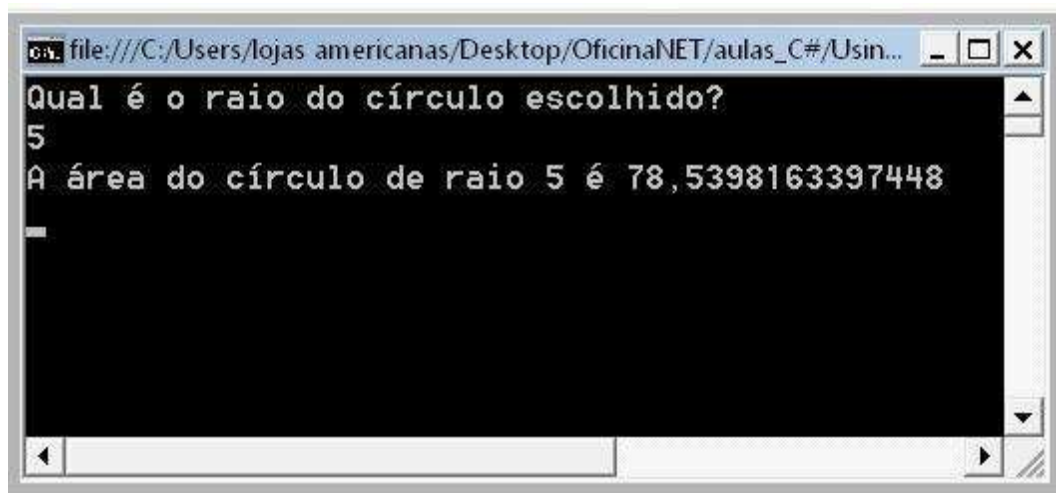


figura 30: resultado da aplicação.

Trabalhando com construtores

Quando você usa a palavra chave **new** para criar um objeto, a execução tem que construir aquele objeto usando a definição da classe. Em execução, o sistema operacional tem que separar um pedaço de memória a partir do sistema operacional, preencher com campos definidos pela classe, e então invoca um **construtor** para realizar qualquer inicialização necessária.

Um construtor é um método especial que é executado automaticamente quando você cria uma instância da classe. Ele tem o mesmo nome da classe, e pode levar parâmetros, mas não pode retornar valor. Se você não escrever um construtor, a IDE criará um para você, embora o construtor assim construído não faça nada.

Para escrevermos o nosso próprio construtor default, escrevemos um **método público** com o mesmo nome da classe, sem valor de retorno. O exemplo abaixo mostra como, apesar de você inicializar aparentemente um valor na construtor default, na verdade ele não fornece valor algum de retorno para o programa principal. Quando o método da classe **Operações** é chamado, este sim, solicita a entrada de valor, o qual é usado para calcular a área. Veja a classe e o programa principal abaixo listados:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Construtor_default_01
{
    class Operação
    {
        int raio;

        public Operação() //construtor default
        {
            int raio = 15;
        }

        public double DefinirArea()
        {
            Console.WriteLine("Defina o raio: ");

            int raio =
Convert.ToInt16(Console.ReadLine());

            return Math.PI * raio * raio;
        }
    }
}

```

Agora, o programa principal instancia um objeto da classe Operações, usando a palavra reservada **new**:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Construtor_default_01
{
    class Program
    {
        static void Main(string[] args)
        {
            Operação valor = new Operação();

            Console.WriteLine(valor.DefinirArea().ToString());

            Console.ReadLine();
        }
    }
}

```



```

    }
}
}

```

O resultado do programa é:

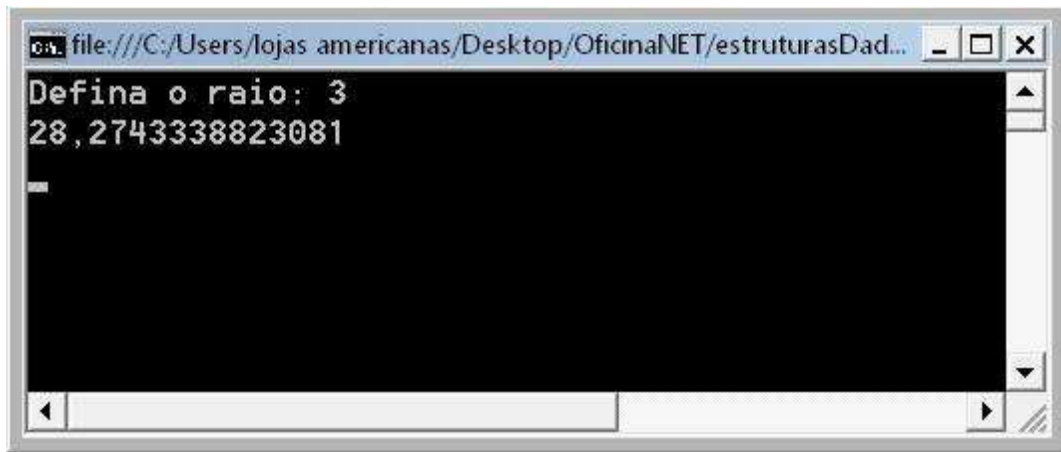


figura 31: entrada e saída de dados (input/output).

Veja que, mesmo tentando inserir o valor 15 no construtor default, o mesmo não passou valor algum ao programa principal, o qual foi solicitado apenas no método da classe.

Sobrecarga de construtores

Quem está habituado a programar em C++ não estranha este termo, a sobrecarga de métodos e construtores tem a mesma função em C#: escrevemos o mesmo nome de método mas usando uma lista de argumentos diferentes (assinatura do método).

Na verdade, construtores são métodos especiais, mas ainda sim são métodos e portanto podem ser sobrecarregados. Veja o exemplo abaixo: construa uma classe chamada **Operações** e escreva o código abaixo:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace sobrecargaCosntrutores
{
    class Operações
    {
        public Operações() //construtos default
        {
            raio = 0;
        }
        //sobrecarga do construtor:
    }
}

```

```

        public Operações(int inicialRaio)
        {
            raio = inicialRaio;
        }

        public double Area()
        {
            return Math.PI * raio * raio;
        }

        private int raio;
    }
}

```

Agora, o programa principal apresenta o seguinte código:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace sobrecargaCosntrutores
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\n Entre raio: ");
            Int32 valorRaio =
Convert.ToInt32(Console.ReadLine());

            Operações círculo = new Operações(valorRaio);

            Console.WriteLine("\n A área do cir de raio {0} e
{1}", valorRaio, círculo.Area().ToString());

            Console.ReadLine();
        }
    }
}

```

A sua execução fornece:

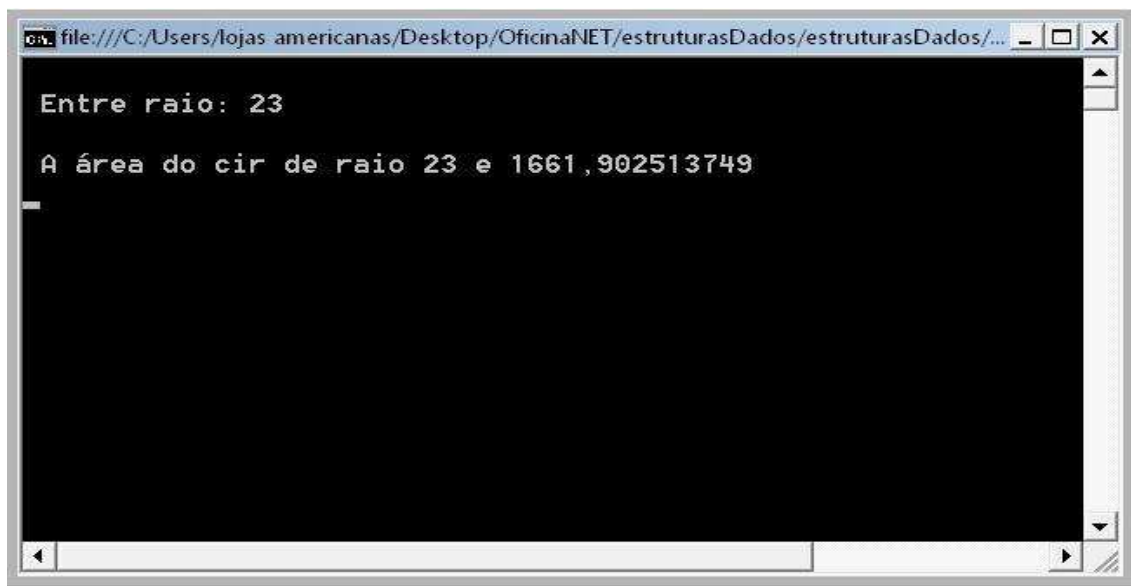


figura 32: execução da aplicação.

Ao compilar o programa, o compilador identifica o tipo de sobrecarga, se ela existe, para a chamada do **construtor**. Como nós passamos um parâmetro ao construtor, o compilador percebe que o mesmo é aquele que recebe parâmetro inteiro e devolve ao programa principal para o cálculo da área. Se não houvesse a sobrecarga, não poderíamos passar parâmetro para a chamada do objeto **círculo** cujo objeto prescinde da palavra **new** para alocar-se em memória.

Compreendendo métodos estáticos e dados

Em C#, todos os métodos devem ser declarados dentro de uma classe. Entretanto, se você declara um método ou campo como estático (**static**) você pode chamar o método ou acessar o campo usando o nome da classe. Nenhuma instância é necessária, isto é, você não precisa da palavra-chave **new** para alocação. Vejamos um exemplo:

Compreendendo valores e referências

Coletivamente, os tipos de dados como `int`, `double` e `char` são tipos de valor, ou tipos valor. Quando declaramos um tipo de valor, o compilador separa um espaço suficiente na memória para armazenar o conteúdo da variável. Como a memória é dividida em células, alguns tipos podem ocupar mais de uma célula consecutiva (lembre-se que a maioria dos PCs usa memórias com células de 8 bits).

Por exemplo, se você declara uma variável tipo `int`, o compilador deverá alocar 4bytes de memória = 32 bits, ou 4 células se a arquitetura das células for de 8 bits.

Um determinado valor declarado como `int` deverá ocupar algum espaço na memória, usando quantas células forem necessárias.

Agora, a memória do computador ainda sofre uma segunda segmentação, ela é dividida em dois setores diferentes: a **Pilha** e o **Heap**. O termo **Heap** não possui correlação na nossa língua de modo que não pode ser traduzido. Os dados armazenados em Heap podem ser recuperados a partir de qualquer localização, de modo que não uma ordem na recuperação dos dados a partir do **Heap**. Por outro lado, o **Stack** armazena os dados de modo que a sua recuperação se dá a partir do último elemento armazenado, isto é, a leitura destes ocorre de acordo com a política: primeiro dado a sair é o último que entrou.

Esta estrutura é semelhante a um pilha de pratos: você vai retirando os pratos a partir do último que foi colocado, até chegar ao primeiro prato, que fica no começo da pilha, embaixo. Na seção sobre estruturas de dados vamos explicar este tipo de estrutura e como é implementada em C#.

Estruturas de Dados

As **estruturas de dados** são coleções de tipos definidos pelo programador e são extremamente úteis no dia a dia do desenvolvedor.

Array é o primeiro tipo de estrutura de dados que devemos estudar, é semelhante a um vetor, e exige a determinação da dimensão do espaço vetorial.

Array

Um array é uma estrutura de dados uniforme, que guarda seus elementos como uma coleção de dados do mesmo tipo. Os elementos individuais de um array são obtidos através da declaração: `MyArray[5]`. Neste caso, obtivemos o sexto elemento do array cujo nome é `MyArray`. Um array precisa da determinação do tipo de dados e a dimensão deste. Há dois tipos fundamentais de arrays em C#: arrays unidimensionais e arrays multidimensionais. Vamos começar estudando o primeiro caso.

Array unidimensional

Esta estrutura é algumas vezes conhecida por vetor ou arranjo. Ela exige a determinação da dimensão da estrutura, que pode ser especificada no programa ou via entrada pelo usuário. Nesta estrutura, o primeiro elemento sempre começa com índice 0.

Há alguns métodos muito importantes que podem ser aplicados sobre a estrutura array, um deles é a propriedade **Length**, que determina ou retorna, a dimensão do array. No exemplo abaixo, escrevemos um loop que escreve componentes inteiras para um vetor e as imprime em tela:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace arranjos_01
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] vetor = new int[5];

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

            Console.ReadLine();
        }
    }
}

```

Podemos usar este tipo de estrutura se um tipo de dado forma uma espécie de sequência de valores de mesmo tipo, por exemplo, a estrutura de dados **funcionários** pode ser um bom exemplo para array, o primeiro elemento **funcionários[0]** é o primeiro funcionário, e assim por diante.

Exemplo console usando um array unidimensional

Vamos escrever um exemplo chamado tipo **Loteria**, no qual nós usamos uma estrutura tipo Array[] e um gerador de números aleatórios, usando a classe **Random** e uma instância da mesma.

O usuário entra com 3 números e em seguida o computador testa com os números por ele gerados. Um acumulador soma os acertos e fornece a premiação.

Acompanhe o código abaixo:

```

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

```

```

namespace Loteria_04
{
    class Program
    {
        static void Main(string[] args)
        {
            TempoLocal tempo = new TempoLocal( );

            #region Variáveis_Principais:
            string retorno;
            Random r = new Random( );
            int[] aposta = new int[3];
            int[] palpito = new int[3];

            //acumulador dos pontos da Loteria, se
            pontos = 3, ganhou prêmio máximo.

            int pontos;
            pontos = 0;

            int caso;
            #endregion
do
{
            Gerando_Números( r, aposta );

            //Aposta do cliente da Loteria da Caixa:
            label:
            Console.WriteLine( "\n Loterias da Caixa: Digite
            três números inteiros entre 1 e 10: " );

            #region Apostas_do_Usuário:
            label2:
            try
            {
                Console.Write( "\n Número 1 = " );
                palpito[0] = Convert.ToInt32(
Console.ReadLine( ) );
                Console.Write( "\n Número 2 = " );
                palpito[1] = Convert.ToInt32(
Console.ReadLine( ) );
                Console.Write( "\n Número 3 = " );
                palpito[2] = Convert.ToInt32(
Console.ReadLine( ) );
            }

```

```

        catch (FormatException ex)
        {
            Console.Write( ex );
            goto label2;
        }
    #endregion

//algoritmo para validar os palpites:

#region Calcula_Apostas:
    if (palpite[0] >= 0 && palpite[0] <= 10)
    {
        if (palpite[1] >= 0 && palpite[1] <= 10 &&
palpite[1] != palpite[0])
        {
            if (palpite[2] >= 0 && palpite[2] <= 10 &&
palpite[2] != palpite[1] && palpite[2] != palpite[0])
            {
                Console.Write( "\n Aposta Válida, gerando
resultado... " );
                //algoritmo de
acertos:
                caso = 1;
                for (int J = 0; J < aposta.Length; J++)
                {
                    switch (caso)
                    {
                        case 1:
                            if (palpite[J] == aposta[0] || palpite[J] ==
aposta[1] || palpite[J] == aposta[2])
                            {

                                pontos = pontos + 1;
                                }
                                break;

                                default:
                                {
                                    }
                                    break;
                                }
                                }
                                Console.Write( "\n Você acertou {0} pontos",
pontos );

                                //fim do algoritmo de acertos.
                            }

```

```

else
{
    Console.Write( "\n Palpite inválido!" );
    goto label;
}
}
else
{
    Console.Write( "\n Palpite inválido!" );
    goto label;
}
}
else
{
    Console.Write( "\n Palpite inválido!" );
    goto label;
}

//Fim da Verificação.
#endregion

Premiação( pontos );

Console.Write( "\n Aposta do dia {0}/{1}/{2}:",
tempo.RetorneDia(),tempo.RetorneMês(),tempo.RetorneYear() );

Console.Write( "\n {0}, {1}, {2}", aposta[0],
aposta[1], aposta[2] );

Console.WriteLine( "\n
*****" );

Console.Write( "\n Deseja refazer outra aposta? (s/n):
" );
retorno = Console.ReadLine();
//anule os pontos no retorno do código:
pontos = 0;
Console.Clear();

} while (retorno == "s" || retorno == "S");

Console.WriteLine( "\n FIM do Programa de Apostas." );

Console.ReadLine();
}

private static void Premiação(int pontos)

```



```

{
    label:
    Random rp = new Random( );
    int premio = rp.Next( ) % 10;
    int premioOferecido;

    if (premio > 0)
    {
        premioOferecido = 10000 * premio;
    }
    else
    {
        goto label;
    }

    #region Premiação:
    switch (pontos)
    {
        case 1:
            Console.Write( "\n Você não ganha nada desta
vez!" );
            break;

        case 2:
            Console.Write( "\n Prêmio = R$ {0}", 0.85 *
premioOferecido );
            break;

        case 3:
            Console.Write( "\n Parabéns: Prêmio
máximo = R$ {0}", premioOferecido );
            break;

        default:
            break;
    }
    #endregion
}

```

```

private static void Gerando_Números(Random r, int[]
aposta)
{

    aposta[0] = r.Next( ) % 10;
    // se repetir ele volta aqui:

```

```

        label:
        aposta[1] = r.Next( ) % 10;

        #region pedaço
        if (aposta[1] == aposta[0])
        {
            goto label;
        }
        aposta[2] = r.Next( ) % 10;
        // a última não pode ser igual à 0 ou 1, senão
        volta ao label:
        if (aposta[2] == aposta[1] || aposta[2] ==
        aposta[0])
        {
            goto label;
        }
        #endregion
    }

    //evita repetições entre os números gerados.
}

```

Escrever aqui o código da classe TempoLocal:

Alguns comentários

Qualquer número aleatório gerado em C# depende de uma semente, que escrevemos como: `Random r = new Random();` para implementarmos o número em si, deveremos escrever `r.Next();` O segmento de código `r.Next()%10;` significa um número entre 0 e 10 e assim por diante. `%` extrai o resto da divisão inteira de dois números. Note que o prêmio também foi gerado por um número aleatório.

Uso do goto

O `goto` não é muito recomendado como boa prática de programação, mas deve-se ressaltar que é importante colocá-lo de modo muito estratégico (se você realmente desejar usá-lo), para que o programa não perca a característica de **estruturação**.

Se você usar num mesmo método mais do que 2 **goto** é provável que o acompanhamento do raciocínio do programa se torne muito difícil. Numa situação de menu de apresentação, onde o usuário pode escolher encerrar o programa, ao invés de ir para outro item, pode ser usado sem perda de estruturação. O que ocorre é que o `goto` deve estar bastante desacoplado para que o mesmo não interfira no fluxo de raciocínio

de modo obscuro, assim ocorrendo, pode ser usado sem problema. Tome cuidado e opte por outras estruturas quando tiver opção.

Array multidimensional

Arrays multidimensionais são objetos mais gerais do que o unidimensional, pois cada elemento do array é um subarray ou um vetor.

Há dois tipos de arrays multidimensionais:

- 1) Arrays retangulares e
- 2) Jagged arrays

Retangulares: São arrays onde todos os subarrays numa dimensão particular têm o mesmo comprimento.

Exemplo: `int x = myArray[4, 6, 2];`

Jagged: São arrays multidimensionais onde cada subarray é um array independente, podem existir subarrays de diferentes comprimentos, e usam um conjunto de colchetes diferentes para cada dimensão do array:

Exemplo: `int x = jagArray[3][4][5];`

Uma instância de um Array é um objeto derivado da classe `System.Array`. Desta forma, há uma série de propriedades que eles herdam desta classe, tais como:

- 1) `Rank` = uma ppdde que retorna o número de dimensões do array.
- 2) `Length` = retorna o número total de elementos do array.

Arrays são **tipos de referência**, logo eles têm ambos, uma referência aos dados e os dados em si (que moram sempre no heap). Agora os dados por sua vez podem ser tipo valor ou tipo referência. Se são tipo valor, são valores armazenados diretamente no heap, se são tipo referência, apontam para outros dados no heap.

Instanciando arrays uni-dimensionais ou retangulares

Para instanciar um array, você usa uma expressão de criação do array, a qual consiste do operador `new`, seguido do tipo de base, e seguido por um par de colchetes. O comprimento de cada dimensão é colocado numa lista separada por vírgulas dentro deste par de colchetes.

Os seguintes são exemplos de declarações de arrays:

- 1) `m_array1` é um array unidimensional de 4 inteiros.
- 2) `m_array2` é um array unidimensional de 4 referências da classe `MyClass`
- 3) `m_array3` é um array 3-dimensional.

```
int[] m_array1 = new int[4];
MyClass[] m_array2 = new MyClass[4];
int[ , , ] m_array3 = new int[3, 6, 2];
```

O **comprimento** do último array é: $3 \times 6 \times 2 = 36$, isto é, pode armazenar 36 elementos do tipo inteiro. O primeiro deles é: `m_array3[0,0,0]`.

Acessando elementos de um array

Um elemento de um array é acessado usando um valor inteiro como índice do array, cada dimensão tem o seu primeiro elemento como rotulado por 0, e o índice é colocado dentro do colchetes. Abaixo mostramos um exemplo de um array cujos valores não são determinados por entrada de dados, mas há uma regra dentro de um laço for para calcular o próximo elemento. O último laço mostra os índices e o valor de cada elemento:

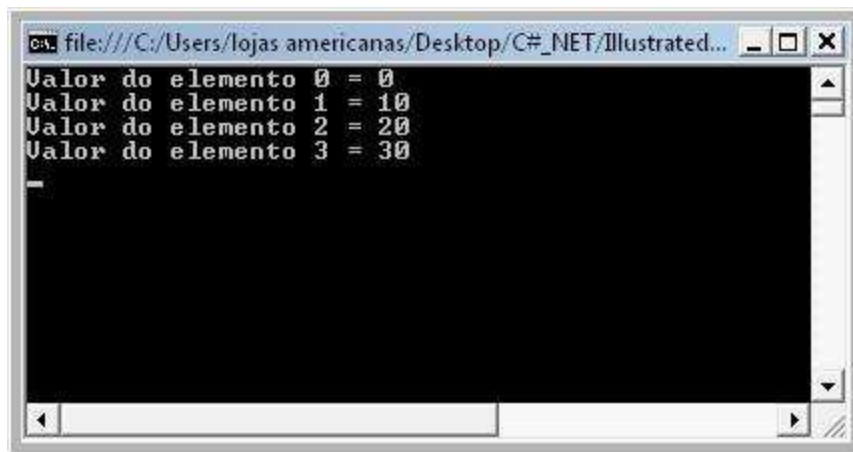
```
static void Main( string[] args )
{
    int[] myIntArray;                                // Declara o array.

    myIntArray = new int[4];                          // Instancia o array.

    for ( int i=0; i < 4; i++ )                      // Determina valores.
    {
        myIntArray[ i ] = i * 10;
    }

    for ( int i=0; i < 4; i++ )                      // Lê e mostra cada elemento.
    {
        Console.WriteLine( "Valor do elemento {0} = {1}",
                            i, myIntArray[i] );
    }
}
```

Sua execução fornece:

A screenshot of a C# console application window. The title bar shows the file path: "file:///C:/Users/lojas americanas/Desktop/C#_NET/Illustrated...". The console output displays the values of an array with four elements: "Valor do elemento 0 = 0", "Valor do elemento 1 = 10", "Valor do elemento 2 = 20", and "Valor do elemento 3 = 30". There is a small horizontal line below the last line of output.

```
file:///C:/Users/lojas americanas/Desktop/C#_NET/Illustrated...
Valor do elemento 0 = 0
Valor do elemento 1 = 10
Valor do elemento 2 = 20
Valor do elemento 3 = 30
-
```

Pilhas

Este tipo de estrutura de dados é implementada pela **classe Stack**. Deve-se determinar o tipo de pilha a ser usada, como: **Stack<T> pilha = new Stack<T>**, onde T é o tipo de dados da pilha. No exemplo que mostraremos abaixo, construímos um programa que muda a base de representação de um número inteiro, da base decimal para uma base qualquer entre 2 a 10.

O tipo de estrutura de dados pilha é extremamente importante na computação, os elementos de uma pilha são guardados na ordem de chegada, mas recuperados na ordem inversa, isto é, o primeiro elemento a ser retirado da pilha foi o último a ser acrescentado na mesma.

É uma tentação usar este tipo de estrutura para elaborarmos um programa para a mudança de base de um número, pois o algoritmo que está por trás desta construção baseia-se nas divisões sucessivas do número dado, pelo divisor que representa a base. Neste algoritmo, começa-se dividindo o número dado pela nova base, armazena-se o quociente e o resto. Este primeiro quociente é novamente dividido pela base, gerando-se os próximos resto e quociente. Isto prossegue até que o quociente seja menor que a base. Para escrevermos o número na nova base, pegamos o último quociente (que é menor que a base) e começamos a escrever os restos, na ordem inversa na qual eles apareceram.

Desta forma, é muito natural que a estrutura de dados pilha, implementada pela classe **Stack**, seja capaz de resolver este problema para nós. Abaixo eu escrevo a minha versão do programa e imprimo algumas capturas de tela para acompanharmos a recuperação de seus dados.

Abram uma aplicação **Console** e copiem e coleem o código abaixo:

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

///<\data> 18/03/2009: Criado por Paulo Sérgio
Custódio,
///<\title> Professor de Sistemas de Informação -
UNIFIG
///<\title>
///<\data>

namespace Codesnippet
{
    class Program
    {
        static void Main(string[] args)
        {
            Int32 quociente, dividendo, divisor, resto;
            Int32 numero; string retorno;

            do
            {
                Console.Write("\n Esta aplicação converte um
número ");
                Console.WriteLine(" digitado na base 10, para a
base binária");

                Console.WriteLine("*****
*****");

                Console.Write("\n Entre número, e logo após uma
base: ");

                try
                {
                    numero = Convert.ToInt32(Console.ReadLine());
```

```

        label:
            Console.WriteLine("\n Digite a nova base: ");
            Int32 novaBase =
Convert.ToInt32(Console.ReadLine());
            divisor = novaBase;
                        //limita os tipos de bases
digitados:
            if (divisor > 10 || divisor <=0)
            {
                Console.WriteLine("\n Programa limitado a bases <=
10 e positivas.");
                goto label;
            }

#region Switch(divisor)

switch (divisor)
{

    case 2:
        Console.WriteLine("\n A base escolhida é a binária. ");
        break;
    case 8:
        Console.WriteLine("\n A base escolhida é a octal. ");
        break;
    case 16:
        Console.WriteLine("\n A base escolhida é a hexadecimal.
");
        break;
    default:
        Console.WriteLine("\n A base escolhida é genérica.");
        break;
}
#endregion

//declaração da pilha:

Stack<Int32> pilha = new Stack<Int32>();

    dividendo = numero;

//algoritmo principal:
do
{
    quociente = dividendo / divisor;
    resto = dividendo % divisor;

```

```

        pilha.Push(resto);
        dividendo = quociente;

    }while (quociente >= divisor);

//calcula o último quociente, que deve aparecer na
listagem:
    pilha.Push(quociente);

Console.WriteLine("\n O número {0} escrito na base {1} e:
", numero, novaBase);

while (pilha.Count > 0)
{
    Console.WriteLine(pilha.Pop().ToString() + "");
}

}
catch (FormatException f)
{
    //mensagens de exceção:
    Console.WriteLine("Message: {0}", f.Message);
    Console.WriteLine("Source: {0}", f.Source);
    Console.WriteLine("Stack: {0}", f.StackTrace);
    //limpeza da tela de erros:

    string resposta;
    Console.WriteLine("\n Deseja limpar a tela de
erros?(s/n) ");
    resposta = Console.ReadLine();

    if (resposta == "s" || resposta == "S")
    {
        Console.Clear();
    }

}
catch { }

//o último catch captura erros genéricos.
//deve ser sempre usado.

Console.WriteLine("\n ");

```



```

Console.WriteLine("\n Deseja repetir cálculos com
outro número?(s/n)");

retorno = Console.ReadLine();

Console.Clear();

}while(retorno == "s" || retorno == "S");

    Console.WriteLine("\n FIM do Programa.");

    Console.ReadLine();

        }
    }
}

```

Vamos executar este programa diversas vezes e acompanhar as suas saídas, e inclusive, vamos forçar números equivocados para analisarmos o tratamento de exceções, determinado pelas cláusulas try catch.

DESAFIO PUTNAM

Refaça o programa acima, implementando-o de forma a calcular a mudança para a base hexadecimal ($B = 16$), note que nesta base, os caracteres são: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. A problemática aqui é a mudança de inteiro para caracter, como você procederia? Em seguida implemente-o para todas as principais bases: 2, 8, 10 e 16, e mostre um aviso ao usuário se a base escolhida não for adequada ou fora de intervalo.

Dica: estude o conceito **objeto**.

Listas, Listasligadas, Dictionary, Queue

Árvores e árvores binárias

Herança

A herança é a capacidade de uma instância de uma dada classe poder herdar propriedades e objetos de outra classe. Para especificarmos a herança de uma classe, precisamos de um projeto com pelo menos duas classes adicionais para ilustrarmos este conceito. Podemos proceder ainda de dois modos para estabelecermos a herança:

- 1) Codificar manualmente ou
- 2) Usar o Diagrama de Classes.

Vamos abordar estes dois métodos aqui, usando um exemplo simples (pois vamos apenas esclarecer o conceito de herança).

Crie um novo aplicativo **Console** em C#, e dê clique com o botão direito do mouse no nome do projeto, para adicionarmos uma classe: Após clicarmos no nome do projeto -> Add New Item deve aparecer a janela:

Dê nome à nova classe como **Carros.cs**:

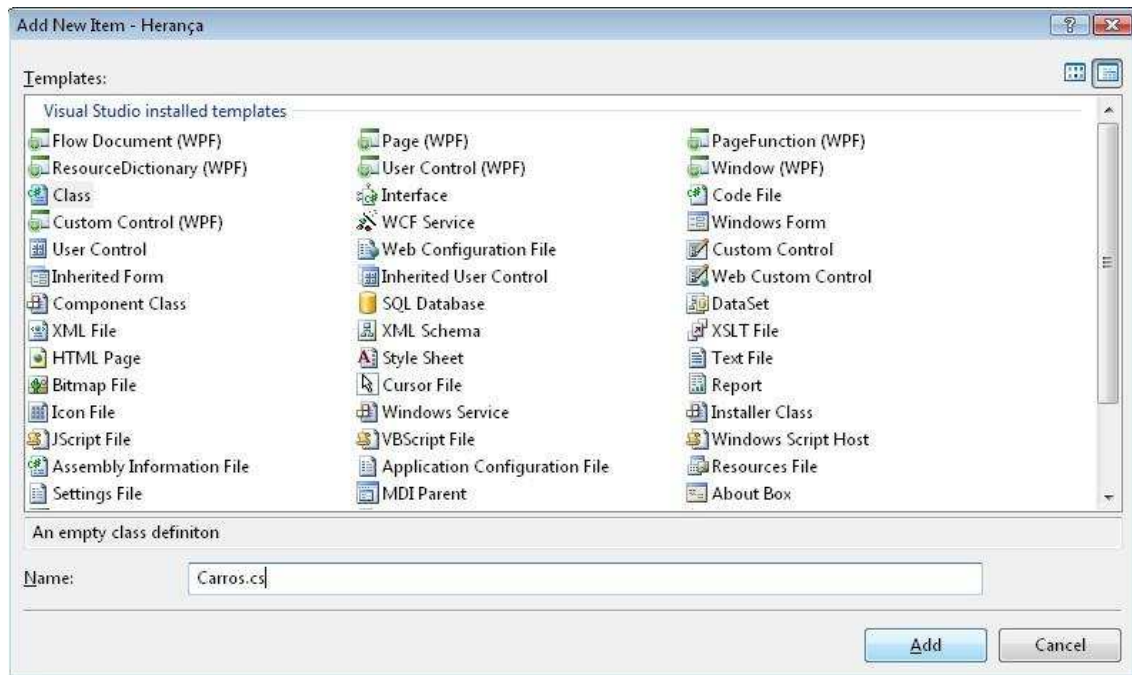


figura 33: definindo a classe.

Clique **Add**, para adicionar esta classe. A nova classe adicionada aparece no Solution Explorer, veja um zoom (apenas desta região):

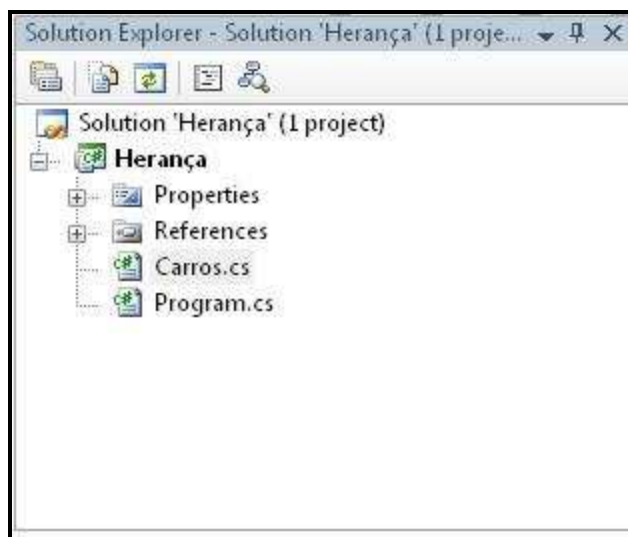


figura : localização dos arquivos do projeto.

Agora, crie mais uma nova classe e a adicione a este projeto, denomine-a Utilitários.cs, o **Solution Explorer** deverá ficar assim:

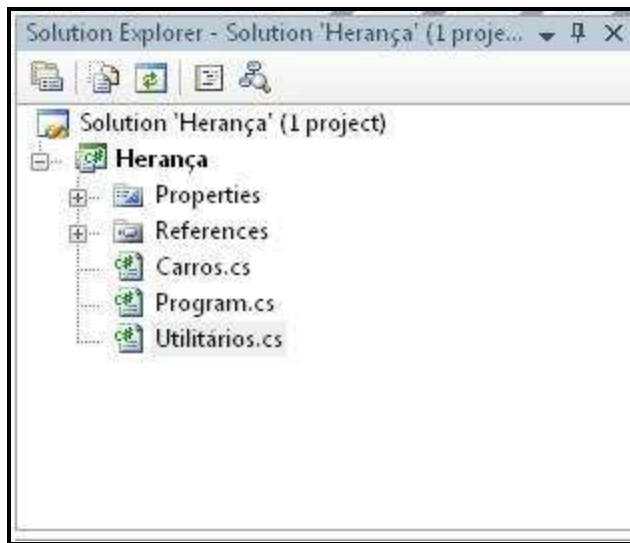


figura 34: adicionando mais uma classe ao projeto.

Agora podemos construir a herança. O conceito de herança em programação segue as mesmas linhas intuitivas que nós conhecemos. Sabemos que todo carro possui propriedades em comum: possuem motor, rodas, volante, estepe, freios, etc. desta forma, estes itens são comuns à maioria dos carros. Embora **Utilitários** sejam carros com layouts especiais, eles ainda são carros e compartilham, isto é, herdam as propriedades da imensa classe Carros.cs.

Logo, **Utilitários.cs** é a classe que herda todas as propriedades e métodos da classe Carros.cs. Para implementarmos a herança acima, volte ao código do arquivo: Utilitários.cs:

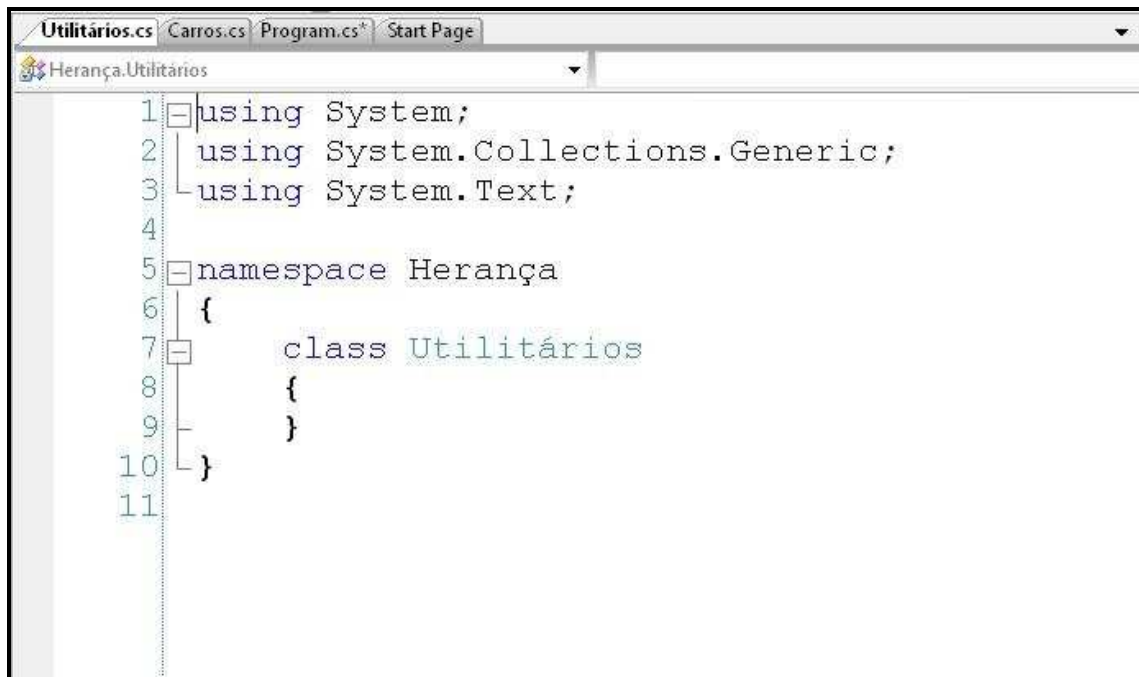


figura 34: abrindo o editor de código na classe Utilitários.cs.

Para implementarmos a herança, escreva após Utilitários: :Carros.cs (incluindo os dois pontos):

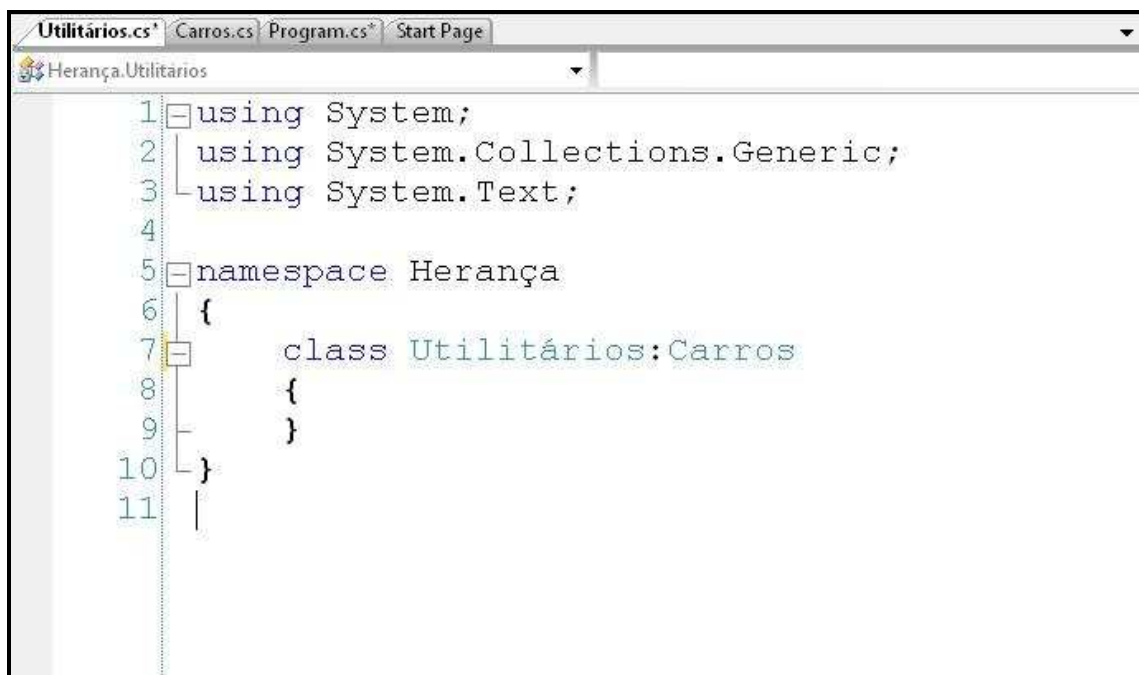


figura 35: implementando a Herança! (abra outro vinho!)

Pronto! A herança entre as duas classes acabou de ser implementada. A classe **Utilitários** herda todas as ppddes da classe Carros. Vamos escrever alguns métodos e propriedades na classe Carros.cs e mostrar como a herança é implementada na prática.

Escreva as seguintes propriedades na classe Carros.cs:

```
class Carros
{
    private int m_pesoMotor;

    public int PesoMotor
    {
        get { return m_pesoMotor; }
        set { m_pesoMotor = value; }
    }

    private int m_velocidadeModelo;

    public int VelocidadeModelo
    {
        get { return m_velocidadeModelo; }
        set { m_velocidadeModelo = value; }
    }

    private string m_modeloCarro;

    public string ModeloCarro
    {
        get { return m_modeloCarro; }
        set { m_modeloCarro = value; }
    }

    //método calcula rendimento aqui.

}
```

Acrescente em seguida, um método para calcularmos o rendimento de um carro qualquer, esta fórmula é dada pelo método:

```
public int Calcula_Rendimento(int peso, int
velocidade)
{
    int m_rendimento;
    m_pesoMotor = peso;
    m_velocidadeModelo = velocidade;

    m_rendimento = m_pesoMotor * m_velocidadeModelo;

    return m_rendimento;
}
```

O método recebe dois argumentos inteiros e retorna o produto dos mesmos (a fórmula de rendimento não é esta, apenas é ilustrativa do que significa um método retornar uma quantidade ou valor), agora vamos ao arquivo **Program.cs** e escreveremos:

```
static void Main(string[] args)
{
    Console.WriteLine("\n Digite peso e vel. máxima do
carro: ");
    int m_massaCarro =
Convert.ToInt32(Console.ReadLine());
    Console.Write("\n Agora, a vel. máxima: ");
    int m_velocidadeMáxima =
Convert.ToInt32(Console.ReadLine());

    //declara a instância de um objeto da classe
    Utilitários (note que não há nenhuma fórmula naquela
    classe):

    Utilitários m_sportage = new Utilitários();

    m_sportage.PesoMotor = m_massaCarro;

    m_sportage.VelocidadeModelo = m_velocidadeMáxima;

    Console.WriteLine("\n Rendimento = {0}",
    m_sportage.Calcula_Rendimento(m_sportage.PesoMotor,
    m_sportage.VelocidadeModelo));

    Console.ReadLine();
}
```

Para percebermos que o objeto da classe **Utilitários.cs** herdou de fato as propriedades e métodos da classe **Carros.cs**, observe o comportamento do **Intellisense** quando terminamos de digitar o nome do objeto seguido de ponto:



figura 36: acessando as propriedades do objeto `m_sportage`, da classe `Utilitários`.

Ao encerrarmos a digitação de `m_sportage.`, aparece um menu Intellisense com todas as opções disponíveis ao objeto: as propriedades **ModeloCarro**, **PesoMotor**, **VelocidadeModelo** e o método **CalculaRendimento** estão disponíveis para o programador. Agora você pode escolher um deles e aplicar.

De fato, o objeto pertence apenas à classe `Utilitários`, pois você o declarou como uma instância daquela classe, e em seguida, os métodos e propriedades que estão declarados na classe `Carros.cs`, aparecem disponíveis ao objeto **`m_sportage`**! Esta é a essência da Herança!

Através da declaração **`class Utilitários : Carros`**, os dois pontos estabelecem a relação de Herança: a classe `Utilitários` vai herdar todos os métodos, campos e propriedades da classe `Carros.cs`.

Note que, gratuitamente, ganhamos com este exemplo uma excelente noção do que é verdadeiramente a orientação a objetos: Vamos dar mais uma olhada no Intellisense que acima foi aberto.

Note que ao objeto `m_sportage`, podem ser aplicados algumas propriedades e um método. As propriedades são justamente as propriedades do **objeto** `m_sportage`. **Veja que o propósito da OOP é dar semântica aos objetos, mesmo que eles sejam entidades verdadeiramente abstratas do mundo da Matemática, não importa, sabemos que objetos possuem atributos, ou seja propriedades.**

A declaração: `m_sportage.PesoMotor` é muito intuitiva ao programador de uma linguagem OOP: é o Peso do Motor do objeto `m_sportage`!

Também repare um conceito muito importante que deve se aplicado de agora em diante: apesar da plataforma .NET possuir os seus objetos pré-definidos e seus tipos básicos

primitivos de suas próprias classes, você não está elaborando um programa realmente orientado a objetos (OOP) se você não implementar as suas próprias classes de funcionalidade.

Isto significa o seguinte, o mesmo algoritmo acima poderia ser escrito da seguinte maneira (escreva dentro do Main num novo projeto, compile e rode):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Digite peso e vel.
máxima do carro: ");
        int m_massaCarro =
Convert.ToInt32(Console.ReadLine());
        Console.Write("agora a vel. máxima: ");

        int m_velocidadeMáxima =
Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("Rendimento
={0}",m_massaCarro*m_velocidadeMáxima);

        Console.ReadLine();
    }
}
```

Este programa tem a mesma execução que o programa anterior, mas não podemos considerá-lo orientado a objetos. Aparentemente, o último programa é mais simples que o primeiro, e de fato é, mas as vantagens da programação OOP (orientada a objetos) são mais claras para programas um pouco mais complexos do que este, onde a orientação a objetos acaba se tornando muito útil.

Note que o programa acima não possui qualquer objeto realmente de alguma classe seja Carros ou Utilitários, logo, em termos de orientação a objetos (apesar de aparecerem objetos internos da linguagem C# nele), não há objetos definidos pelo usuário, há apenas alguns parâmetros formais: massa e velocidade e uma relação de produto entre elas, mais nada.

Considerações sobre a OOP

Alguns exemplos de programas das seções futuras não são orientados a objetos (OOP), com o sentido de o aluno assimilar apenas os conceitos algorítmicos de soluções, convidamos os discentes a converterem estes mesmos programas não orientados a objetos desta apostila em programas verdadeiramente orientados a objetos. Fazendo este

exercício de modo continuado é a única maneira do programador assimilar o conceito da orientação a objetos, e não apenas teorizar sobre OOP!

Há outras linguagens que misturam orientação a objetos com orientação funcional, como a F#, e outras voltadas à programação lógica como a Prolog. Sugiro a leitura do excelente livro de Sebesta, R. Introdução aos conceitos de linguagem de programação, para você se aprofundar sobre as outras arquiteturas de linguagem.

Questão para o aluno: Por que usamos o prefixo `m_` antes de `sportage` e das demais variáveis, o `m` seguido de `underscore`? (será frescura?)

Dica: Pense em programas grandes com mais de 40 variáveis e campos e no Intellisense...(você encontrará a resposta).

Mais alguns aplicativos tipo Console

Acessando Serviços do Sistema Operacional

Como exemplo **Console** mais interessante para analisarmos, vamos mostrar uma aplicação que acessa o tempo local do SO e mostra para o usuário se é tarde, noite ou manhã, de acordo com a informação recuperada. Uma vez que os nossos exemplos console serão um pouco mais extensos em linha de código, não vou fazer o print screen da tela, usaremos este recurso apenas em seções curtas. Crie um novo projeto console e uma classe a qual denominaremos `MyClass`. O **Project Explorer** terá o seguinte aspecto:

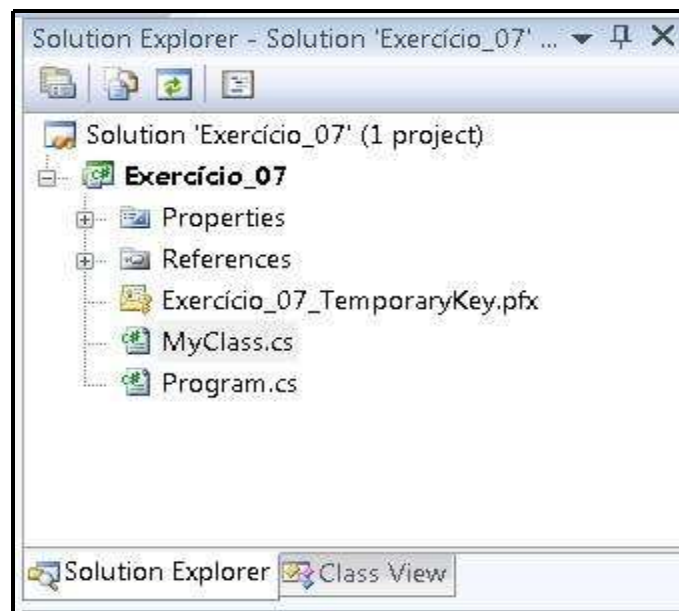


figura 37: o solution explorer.

Dê duplo clique em `MyClass` no project explorer e construa os seguintes métodos públicos abaixo (todo o código terá o seguinte conteúdo abaixo):

```

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

namespace Exercício_07
{
    class MyClass
    {
        //Métodos públicos que fornecem a Hora e
        Minutos:

        public int RetorneHora( )
        {
            DateTime dt = DateTime.Now;
            int hour = dt.Hour;
            return hour;
        }

        public int RetorneMinutes( )
        {
            DateTime dt = DateTime.Now;
            int minutes = dt.Minute;
            return minutes;
        }
    }
}

```

Os métodos acima retornam hora e minutos para o tempo local, e envia o retorno ao programa principal da classe Program (veja no project explorer). O código de Program é: (Os nossos códigos copiados não possuem boa indentação pois não estão escritos na tela do compilador, mas não se esqueça de arrumar a indentação).

```

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

namespace Exercício_07
{
    class Program
    {
        static void Main(string[] args)
        {

```

```

        MyClass tempo = new MyClass( );
        string retorno;

        do{

            Console.WriteLine("\n *** Aplicação p/ Calcular o
Tempo Local: ***");

            Calcula_Tempo_Horas_plus_Minutos(tempo);

            Console.WriteLine("\n Deseja recalcular o tempo
local? (sim/não)");
            retorno = Console.ReadLine( );

        }while(retorno == "sim" || retorno == "SIM");

        Console.WriteLine("\n FIM do Programa.");

        Console.ReadLine( );
    }

    private static void
    Calcula_Tempo_Horas_plus_Minutos(MyClass tempo)
    {
        Console.WriteLine("\n Hora local é (hh:mm): {0}:{1}
", tempo.RetorneHora( ), tempo.RetorneMinutes( ));

        Decide_Período(tempo);
    }

    private static void Decide_Período(MyClass tempo)
    {
        if (tempo.RetorneHora( ) > 12 &&
tempo.RetorneHora( ) < 18)
        {
            Console.WriteLine("\n É tarde.");
        }
        else
        {
            if (tempo.RetorneHora( ) > 18)
            {
                Console.WriteLine("\n É noite.");
            }
            else

                if(tempo.RetorneHora( ) < 12)
                {

```

```

        Console.WriteLine("\n É manhã");
    }

}

//fim do método.
}

}

}

```

Comentários sobre o código acima

O método **public int RetorneHora()** contém um objeto dt da classe **DateTime**. Esta classe fornece suporte aos eventos do tempo do sistema. Uma variável desta classe pode capturar o tempo até em milissegundos. Com a propriedade **.Now**, extraímos o tempo presente e na segunda linha aplicamos a propriedade **.Hour**, para podermos capturar a hora presente. Da mesma maneira, o método **RetorneMinutos()** retorna o minuto presente através da propriedade **.Minute**, aplicada ao objeto dt, da classe **DateTime**.

Em seguida, na classe **Program**, a qual contém o método principal (**Main**), criamos uma instância da classe **MyClass**, chamada **tempo**. Sobre esta instância podemos rodar dois métodos da classe **MyClass**, para extraírmos o tempo presente em hora e minutos.

A variável tipo **string** **retorno** permite o retorno do programa como um todo, através do **do { }...while();** de acordo com a resposta do usuário.

O método **private static void Calcula_Tempo_Horas_plus_Minutos(MyClass tempo)** escreve o tempo presente em termos de horas e minutos usando o método **Writeline** da classe **Console**. Note que **tempo.RetorneHora()** e **tempo.RetorneMinutos()** são os ingredientes que executam esta tarefa, chamando a instância da classe **MyClass** e atuando sobre este objeto dois métodos daquela classe, um para obter a hora e o outro os minutos. Em seguida, dentro deste método nós escrevemos o método: **Decide_Período(tempo)**. Ele usa uma estrutura de seleção **if ... else** para decidir o status do período.

A execução deste aplicativo fornece o tempo local numa janela console e decide o status do período:

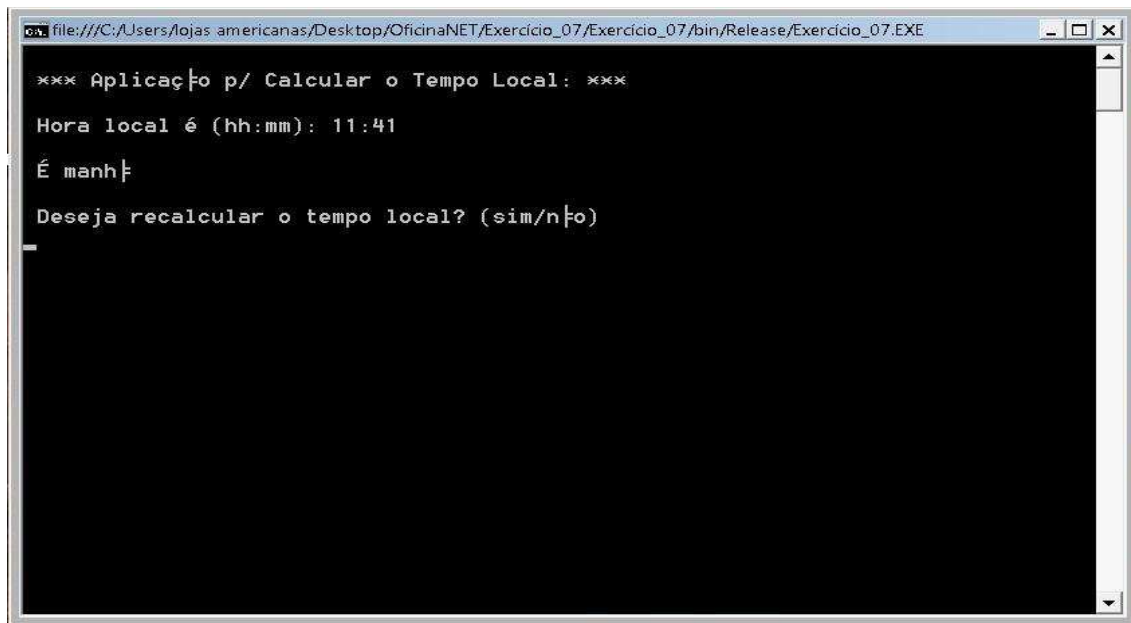


figura 38: execução do programa de captura do tempo local.

Como vimos, a classe **DateTime** permite a comunicação entre o aplicativo console e o tempo que é continuamente atualizado no sistema.

Menu de opções

Quero um aplicativo console com um menu de 2 opções: a primeira opção deve permitir que o usuário calcular a raiz quadrada de um número e a segunda opção converte o número digitado para a base 2. O menu de opções deve se parecer com a figura abaixo:

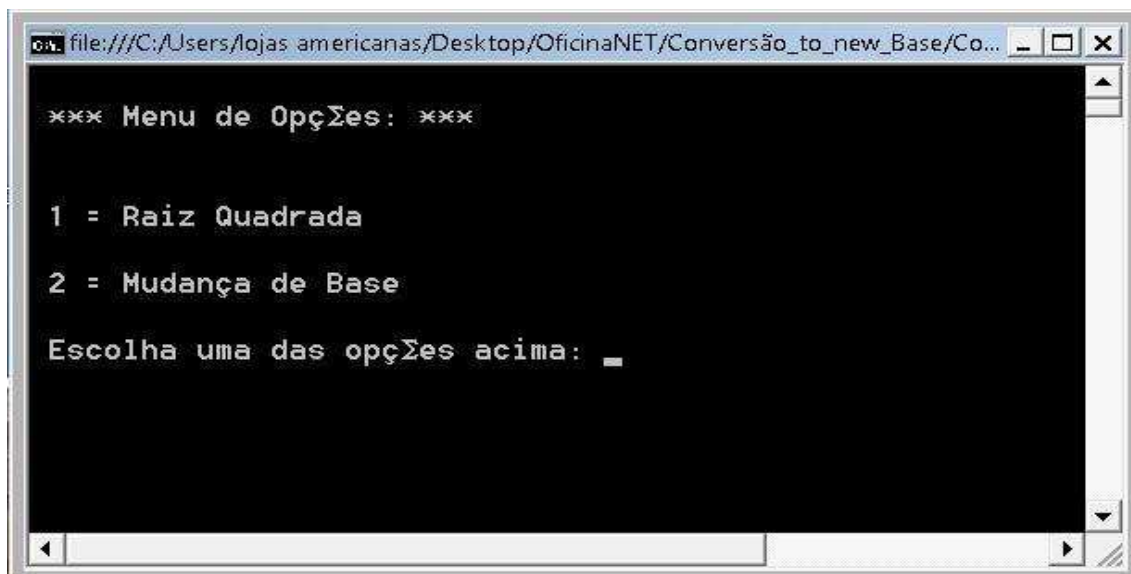


figura 39: menu de opções.

Este aplicativo exemplifica o uso das classes de exceção (tratamento de erro) que são comuns em todos os aplicativos profissionais (sejam de qualquer natureza), o

importantíssimo recurso de desvio de fluxo goto, e a estrutura de seleção switch case (mais elegante e eficiente que a cláusula if else (ou if then else em outras linguagens).

Crie um novo projeto console e o denomine Conversão_to_new_Base.

Feito isto adicione uma classe ao seu projeto, a qual denominar-se-á Conversão.cs. Lembre-se que para adicionarmos uma classe ao projeto basta selecionarmos o nome do projeto e com o botão direito do mouse vamos ao item Add -> Class:

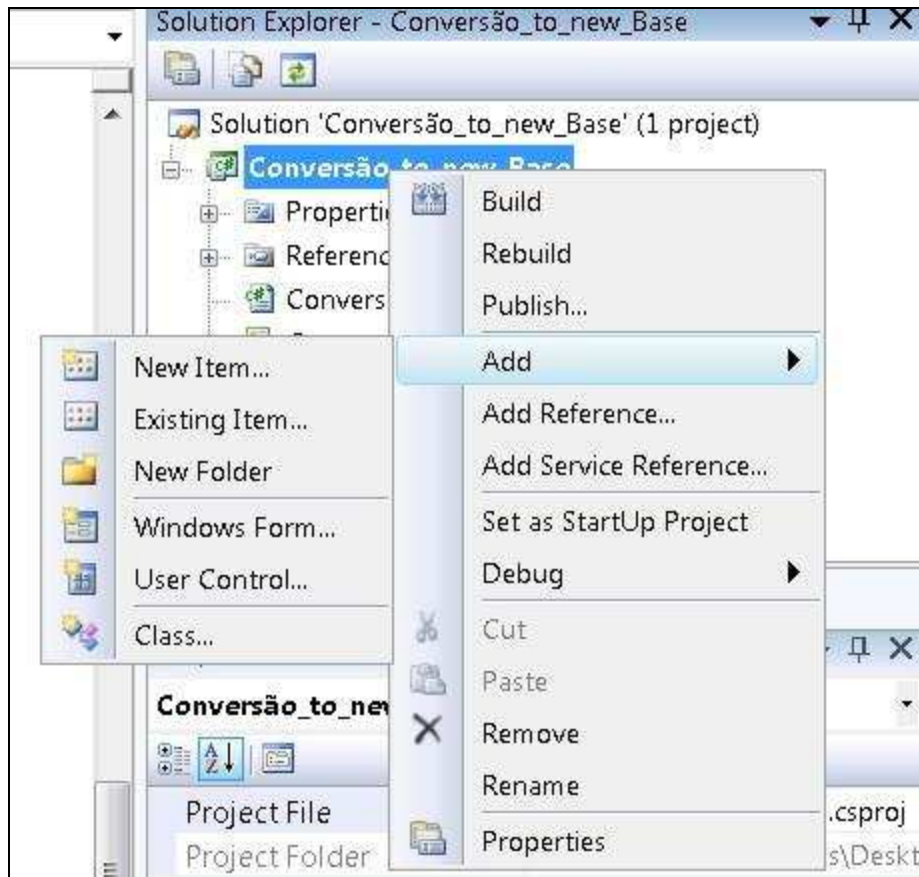


figura 40: adicionando uma classe ao projeto

Denomine esta classe Conversão.cs e escreva o seguinte código para ela:

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

//Classe que executa a conversão em si:

namespace Conversão_to_new_Base
{
    class Conversão
    {
        public Int32 N;
```

```

        public Int32 novaBase;
        public Int32 quociente;

        public void ConversãoNumero(Int32 dividendo,
Int32 divisor)
        {
            N = dividendo;
            divisor = novaBase;
            //some + 1 na dimensão para não faltar a
última divisão:
            Int32[] resto = new
Int32[Convert.ToInt32(Math.Log(N, novaBase)) + 1];

            Console.WriteLine("\n O número {0} na base {1} e:
", N, novaBase);

            for (int J = 0; J <= Math.Log(N, novaBase); J++)
            {
                quociente = dividendo / divisor;
                resto[J] = dividendo % divisor;
                dividendo = quociente;
            }

            //escrevendo de trás p frente:
            for (int J = 0; J <= Convert.ToInt32(Math.Log(N,
novaBase)); J++)
            {
                Console.WriteLine(resto[Convert.ToInt32(Math.Log(N,
novaBase)) - J]);
            }
        }

        public void CalculaRaizQuadrada(Int32 numero)
        {

            Console.WriteLine("\n A raiz quadrada de {0} e {1}",
numero,    Math.Sqrt(numero));

            //fim do método.
        }

    }
}

```

Refaça a indentação deste programa na sua máquina para ficar melhor do que acima!

Há dois métodos muito simples nesta classe, mas o primeiro deles exige um pouco de atenção. Iniciamos esta classe declarando três campos públicos que serão consumidos na classe principal do programa: N, novaBase e quociente. O loop:

```

for (int J = 0; J <= Math.Log(N, novaBase); J++)
{
    quociente = dividendo / divisor;
    resto[J] = dividendo % divisor;
    dividendo = quociente;
}

```

é responsável por executar a rotina das divisões sucessivas do número digitado, isto gera a representação de zeros e uns que desejamos. Note que acumulamos o resto da divisão num vetor: resto[J] de componente – J. Por que fazemos isto? Lembre-se que o algoritmo das divisões sucessivas exige que a representação binária do número seja lida de trás para frente: do último resto em direção ao primeiro. Se tivéssemos acumulado estes restos numa variável não vetorial, não teríamos como recuperar os dados na ordem que desejamos. Isto se deve ao fato de que um vetor armazena os seus dados na pilha e não na fila. O loop abaixo permite a recuperação e escrita dos restos na ordem que o algoritmo necessita:

```

for (int J = 0; J <= Convert.ToInt32(Math.Log(N,
novaBase)); J++)
{
    Console.Write(resto[Convert.ToInt32(Math.Log(N,
novaBase)) - J]);
}

```

O segundo método público simplesmente calcula a raiz quadrada de um número:

```

public void CalculaRaizQuadrada(Int32 numero)
{
    Console.WriteLine("\n A raiz quadrada de {0} e
{1}", numero, Math.Sqrt(numero));
}

```

Os métodos acima serão chamados pelo programa principal e o primeiro deles é chamado de acordo com a escolha do menu pelo usuário. Vamos ao código do método Main:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Numeric;
using System.Reflection;

// Escolha Raiz ou Base 2:

namespace Conversão_to_new_Base

```



```

{
    class Program
    {
        static void Main(string[] args)
        {
            string resposta;

            #region Cálculo_NewBase
do
            {
                Calcula_Conversão_N();

                Console.WriteLine("\n Deseja repetir a
operação com outro número? (sim/não)");

                resposta = Console.ReadLine();

            } while (resposta == "sim" || resposta == "SIM");

            #endregion

            Console.WriteLine("\n FIM do Programa. \n");

            Console.ReadLine();
        }

private static void Calcula_Conversão_N()
{
            Conversão n = new Conversão();

            Console.WriteLine("\n *** Menu de
Opções: ***\n");
            Console.WriteLine("\n 1 = Raiz
Quadrada");
            Console.WriteLine("\n 2 = Mudança de
Base");

            Int32 escolhaMenu;
            try
            {
                label3:
                Console.Write("\n Escolha uma das opções
acima: ");
                escolhaMenu =
Convert.ToInt32(Console.ReadLine());

                switch (escolhaMenu)

```

```

        {
        case 1:
            InputInteiro(n);
            n.CalculaRaizQuadrada(n.N);
            break;

            case 2:
                InputInteiro(n);
                ConversãoNovaBase(n);
            break;
        default:
            Console.WriteLine("\n Digite opção
válida");
            goto label3;
            break;
        }

        catch (FormatException e)
        {
            Console.WriteLine("\n\n Exceção de
formato: {0}", e);
        }
    }

private static void InputInteiro(Conversão n)
{
    Console.WriteLine("\n Digite inteiro: ");

    try
    {
        n.N = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex);
    }
}

private static void ConversãoNovaBase(Conversão n)
{
    label:
        Console.WriteLine("\n Digite a nova base para
realizar a conversão (Base < 10): ");

    try
    {

```

```

        n.novaBase = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex);
    }

    //Chamada da função que realiza a
    conversão p/ a nova base.

    if (n.novaBase <= 10 && n.novaBase > 0)
    {
        n.ConversãoNumero(n.N, n.novaBase);
    }
    else
    {
        Console.Write(" \n Warning!: Base deve ser
menor que 10 e positiva. \n");
        goto label;
    }

    }
}
}

```

Comentários sobre a estrutura do programa

Note que o menu de opções está contido no seguinte bloco

```

Int32 escolhaMenu;

try
{
    label3:
    Console.Write("\n Escolha uma das opções acima: ");
    escolhaMenu = Convert.ToInt32(Console.ReadLine());

    switch (escolhaMenu)
    {
        case 1:
            InputInteiro(n);
            n.CalculaRaizQuadrada(n.N);
            break;
        case 2:
            InputInteiro(n);
            ConversãoNovaBase(n);
            break;
    }
}

```

```
default:
    Console.WriteLine("\n Digite opção válida");
    goto label3;
break;
```

...

O usuário digita 1, 2 ou nenhuma dessas opções, então o fluxo do programa é direcionado para os cases correspondentes ou para default (caso em que ele tenha digitado outro número). O desvio de fluxo **goto label3**; redireciona o fluxo do programa de volta lá para o início deste bloco, permitindo que o usuário restabeleça a escolha.

Caso o usuário não digite um número (isto pode ocorrer e é muito comum a ocorrência de erro de digitação) a diretiva **try ...catch** captura esta exceção e não permite que o seu programa trave. Todo aplicativo profissional (seja Windows Forms, Web Application, etc) usa de modo extremamente freqüente os tratamentos de erro. Isto ocorre devido ao fato de que o sistema está preparado para receber dados que já estão endereçados na memória e os seus tipos já estão pré-definidos. Quando o usuário inadvertidamente digita dados de tipos inconsistentes, o sistema deve capturar (catch{ ... }) todas as exceções possíveis.

Notas importantes:

1) Não há apenas um **catch** para cada **try**. As variáveis a serem tratadas numa ocorrência de exceção devem capturar todas as exceções que possivelmente venham a ocorrer.

2) Quando tratamos as exceções, devemos levar em conta o posicionamento correto das variáveis que serão tratadas, isto é, se elas estão devidamente englobadas dentro de try {...} catch{...}, caso contrário, tal exceção não será tratada, veja o seguinte exemplo mais simplificado abaixo de outro programa simples (o seu objetivo é calcular a divisão de dois inteiros entrados pelo usuário):

```
int num, num2, divisão;
num = Convert.ToInt32( Console.ReadLine( ) ); //Não
está sendo tratada! (fora de try)//

try
{

    num2 = Convert.ToInt32( Console.ReadLine( ) );
    divisão = num / num2;
    Console.WriteLine( "A divisão de {0} por {1} e {2}",
        num, num2, divisão);

}
```

```

catch (FormatException ex1)
{
    Console.Write(ex1);
}
catch (DivideByZeroException ex2)
{
    Console.Write(ex2);
}

catch (OverflowException ex3)
{
    Console.Write(ex3);
}

```

Aconselha-se a escrever uma última cláusula catch sem parâmetros, tipo **catch{}**, de modo que segure alguma última possibilidade de ocorrência de exceção.

Como a fórmula do nosso algoritmo envolve a divisão de dois números, em princípio o usuário pode se distrair e digitar 0 para o segundo número, desta forma, devemos tratar a exceção de divisão por zero. O **Intellisense** encontra para nós a classe `DivideByZeroException`, bastando para isso começar a digitar D... dentro da lista de argumentos do segundo catch acima. Deve haver o catch que captura exceção de formato também, pois além de uma possível divisão por zero, o usuário pode entrar números com letras por erro de digitação ou número muito grande fora do escopo de int como foram declarados. Como saber quantos catch, ou seja, quantas capturas de exceção precisaremos?

É simples, você deve testar o seu programa contra todos os tipos de entrada forçando o seu programa ao limite. Nunca você deve supor que o seu programa está livre de erros de inconsistência de dados numa primeira abordagem. No exemplo acima ainda falta tratarmos a exceção por overflow que deve ocorrer quando o usuário digita números muito grandes (note que declaramos variáveis tipo int, se tivéssemos declarado variáveis tipo double este tipo de exceção fica adiado para números muito maiores).

Consulte o **Help** do **MSDN** para a linguagem C# para você verificar quantos bits cada tipo pré-definido pode alocar na memória. No **Visual Studio 2005** você pode selecionar um item qualquer do código e teclar F1 para abrir o Help do Visual Studio sobre o item que você deseja pesquisar.

E finalmente, note que a variável num não está sendo tratada pelo código de tratamento de erro, ela está fora da cláusula try. Isto não deve ocorrer, você precisa copiar e colar esta declaração dentro de try para fazer o seu tratamento correto. Finalmente, a fórmula que executa a divisão também deve estar dentro de **try**.

Não vamos alongar a nossa discussão além dessa introdução mas ainda falta a diretiva `finally` que geralmente acompanha o tratamento de exceção em C#, mas o tratamento acima funciona. Vamos discutir `finally` em outras ocasiões.

Vamos terminar a nossa descrição do programa principal. Note que o método `InputInteiro(n)` contém tratamento de erro pois o usuário pode digitar números com letras ou simplesmente letras por descuido. O segmento de código que chama os métodos **`CalculaRaizQuadrada(n.N)`** e **`ConversãoNovaBase(n)`** está contido no menu de opções.

```
case 1:
    InputInteiro(n);
    n.CalculaRaizQuadrada(n.N);
break;

case 2:
    InputInteiro(n);
    ConversãoNovaBase(n);
break;
```

Ainda falta aprimorarmos um pouco mais o tratamento de exceções de `InputInteiro(n)` que chama o número digitado pelo usuário para ser calculado a sua raiz ou feita a sua conversão. Mas, como o programa está modularizado e componentizado, estes aprimoramentos podem ser feitos a posteriori, sem prejuízos para o programa como um todo. Aí estão os benefícios da Refatoração quando criamos métodos encapsulados e prestamos atenção às boas práticas de programação, a sua manutenção e aprimoramento ficam mais fáceis de serem implementadas. O método `InputInteiro(Conversão n)` trata o erro com relação à exceção de formato e não com relação a um possível overflow:

```
private static void InputInteiro(Conversão n)
{
    Console.WriteLine("\n Digite inteiro: ");
    try
    {
        n.N = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex);
    }
    //fim do método.
}
```

A modificação é muito simples e basta acrescentarmos este tratamento, o método fica assim:

```
private static void InputInteiro(Conversão n)
{
    Console.WriteLine("\n Digite inteiro: ");
    try
    {
        n.N = Convert.ToInt32( Console.ReadLine( ) );
    }
    catch (FormatException ex)
    {
        Console.WriteLine( ex );
    }
    catch (OverflowException ex2)
    {
        Console.WriteLine( ex2 );
    }

    //fim do método.
}
```

Para finalizar, vamos comentar o método ConversãoNovaBase(Conversão n). Aqui está o seu código:

```
private static void ConversãoNovaBase(Conversão n)
{
    label:
    Console.WriteLine("\n Digite a nova base para realizar a
    conversão: ");

    try
    {
        n.novaBase = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex);
    }

    //Chamada da função que realiza a conversão p/ a
    nova base.

    if (n.novaBase <= 10 && n.novaBase > 0)
    {
        n.ConversãoNumero(n.N, n.novaBase);
    }
}
```

```

    }
else
{
    Console.WriteLine(" \n Warning!: Base deve ser menor
que 10 e positiva. \n");
    goto label;
}
//fim do método.
}

```

O seu significado é simples: se o usuário escolheu o menu de opções para conversão à base 2, ele digita em seguida o número que deve ser convertido e logo após o programa solicita a base. Se a base solicitada estiver num intervalo aceitável (maior que 0 e menor que 10) o programa realiza a conversão, em caso contrário ele volta ao início do código, usando **goto label**; (após mostrar uma frase de advertência). Ao retornar ao início teremos novamente:

```

label:
Console.WriteLine("\n Digite a nova base para realizar a
conversão: ");

```

Logo, o programa solicita uma nova base até que seja digitado um número razoável. Poderíamos aqui comunicar o escopo:

```

Console.WriteLine("\n Digite a nova base para realizar a
conversão (Base < 10): ");

```

Cuja mensagem informa mais sobre o escopo da variável. Agora, vamos falar mais sobre o importante comando que realiza o desvio de fluxo: goto. No código nós usamos o desvio goto dentro do else, o qual força uma saída até que a condição adequada de entrada de dados se cumpra. O fluxograma deste método fica bastante razoável com ele. Entretanto, como boa prática de programação nunca use o desvio de fluxo goto de modo excessivo em seus programas.

Se você usar muitos **goto** dentro de um método em particular, desviando o seu fluxo de lá para cá, você estará incorrendo em risco de escrever um código extremamente nebuloso, obscuro e de difícil manutenção posterior. Isto ocorre devido ao número de rastreamentos e bifurcações lógicas que você estará introduzindo.

Note que, como o método acima é uma caixa preta com relação ao **Main** que é programa principal em última análise, este desvio de fluxo é como se não existisse, portanto, em termos do Main, você pode ter milhares de desvios de fluxo comandados por goto, desde que cada grupo de desvio não esteja em número excessivo dentro de cada método particular.

Aconselha-se não mais que 3 gotos num método e que cada método não possua mais que 10 funcionalidades diferentes. Divirta-se com a execução desta aplicação console e implemente-a numa aplicação visual correspondente. Use tratamento de erro e teste o seu programa contra todas entradas inválidas possíveis. Mãos à obra! Agora vamos escrever algumas aplicações visuais tipo **Windows Forms**.

Compilação condicional

A compilação condicional é um recurso interessante quando você deseja pular a compilação de uma determinada seção do seu código, que por algum motivo, não passava pela compilação usual. Neste caso, a estratégia é você declarar uma variável que marque uma diretiva de compilação condicional. No exemplo abaixo nós declaramos 4 variáveis para compilação condicional e usamos a primeira. Por default, a variável é considerada **true**. Vamos acompanhar o exemplo. Sem a diretiva de compilação condicional, a expressão `Console.WriteLine();` evidentemente não passa pela compilação tradicional pois a IDE deve acusar erro de compilação. Sabemos que a construção `Console.WriteLine("Olá mundo");` passa pela compilação usual pois é sintaticamente correta.

Usando a cláusula **a** com a estrutura abaixo definida, temos a seguinte estrutura: como a expressão **!a** nega o conteúdo true de a, ela é falsa, então o if pula para o else e não executa a compilação da expressão `Console.WriteLine();` logo compilamos condicionalmente o nosso código.

```
#define a

using System;
using System.Collections.Generic;
using System.Text;

namespace Compilação_condicional
{
    class Program
    {
        static void Main(string[] args)
        {

            string retorno;
            do
            {
                int num = 13;
                int par = 12;

                #if !a
                    Console.WriteLine();
                #endif
            }
        }
    }
}
```

```

        #else
            Console.Write("Olá mundo.");
        #endif

        Console.WriteLine("Deseja retornar?(s/n)");
        retorno = Console.ReadLine();
        Console.Clear();

    } while (retorno == "s" || retorno == "S");

    Console.WriteLine("\n FIM do Programa.");
    Console.ReadLine();

}
}
}

```

Se clicarmos em **Build**, o compilador não acusará erros devido à cláusula `!a` em `#if`. Deste modo, este programa pode ser executado e em tela aparece a frase **Olá mundo**.

Esta opção é muito interessante quando você está trabalhando com alguns métodos e modificando-os, acrescentando funcionalidades. Vamos supor que um deles comece a travar com erros de compilação. Você pode prosseguir a compilação e alteração dos demais métodos que estão funcionando adequadamente, até que um pouco mais tarde você consiga resolver a compilação do primeiro método que estava com problemas. A estratégia é introduzir uma diretiva de compilação condicional, como fizemos acima, e inserí-la numa condição `#if !a`, do tipo acima. Você pode continuar com a compilação das demais partes de seu programa até que você consiga corrigir a depuração da seção anterior que estava com problemas, sabendo que os demais métodos foram compilados e depurados adequadamente.

Finalmente, você faz um backup das últimas alterações e retira as cláusulas de compilação condicional quando tiver resolvido o seu problema de depuração numa seção crítica. Consulte outros livros mais completos para conhecer as demais diretivas de processador que estão presentes em C#, pois esta discussão não se encerra apenas com a construção acima. É evidente que o caso acima é supersimplificado usando uma escrita que sabemos dar em erro evidente de sintaxe, mas o objetivo é usarmos estas construções numa situação muito mais complexa do que a que foi acima delineada para efeitos de didática.

