

TRADUÇÃO DA SEGUNDA EDIÇÃO

C# 3.0

Guia de Bolso

*Joseph Albahari &
Ben Albahari*



Rio de Janeiro. 2008

C# 3.0 GUIA DE BOLSO

Do original C# 3.0 Pocket Reference, Third Edition, Copyright © 2008 da Editora Alta Books Ltda. Authorized translation from English language edition, entitled C# 3.0 Pocket Reference, Third Edition, by Joseph Albahari & Ben Albahari, published by ©2008 by O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc, the owner of all rights to publish and sell the same. PORTUGUESE language edition published by Editora Alta Books, Copyright © 2008 by Editora Alta Books.

Todos os direitos reservados e protegidos pela Lei 5988 de 14/12/73. Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônico, mecânico, fotográfico, gravação ou quaisquer outros. Todo o esforço foi feito para fornecer a mais completa e adequada informação, contudo a editora e o(s) autor(es) não assumem responsabilidade pelos resultados e usos da informação fornecida. Recomendamos aos leitores testar a informação, bem como tomar todos os cuidados necessários (como o backup), antes da efetiva utilização. Este livro não contém CD-ROM, disquete ou qualquer outra mídia.

Erratas e atualizações: Sempre nos esforçamos para entregar a você, leitor, um livro livre de erros técnicos ou de conteúdo; porém, nem sempre isso é conseguido, seja por motivo de alteração de software, interpretação ou mesmo quando alguns deslizes constam na versão original de alguns livros que traduzimos. Sendo assim, criamos em nosso site, www.altabooks.com.br, a seção Erratas, onde relataremos, com a devida correção, qualquer erro encontrado em nossos livros.

Avisos e Renúncia de Direitos: Este livro é vendido como está, sem garantia de qualquer tipo, seja expressa ou implícita.

Marcas Registradas: Todos os termos mencionados e reconhecidos como Marca Registrada e/ou comercial são de responsabilidade de seus proprietários. A Editora informa não estar associada a nenhum produto e/ou fornecedor apresentado no livro. No decorrer da obra, imagens, nomes de produtos e fabricantes podem ter sido utilizados, e desde já a Editora informa que o uso é apenas ilustrativo e/ou educativo, não visando ao lucro, favorecimento ou desmerecimento do produto/fabricante.

Produção Editorial: Editora Alta Books
Coordenação Editorial: Thalita Aragão Ramalho
Tradução: Laura Ramos
Revisão: Fernanda Rigamont
Revisão Técnica: Thomas Edson
Diagramação: Claudio Frota

Impresso no Brasil

O código de propriedade intelectual de 1º de Julho de 1992 proíbe expressamente o uso coletivo sem autorização dos detentores do direito autoral da obra, bem como a cópia ilegal do original. Esta prática generalizada nos estabelecimentos de ensino, provoca uma brutal baixa nas vendas dos livros a ponto de impossibilitar os autores de criarem novas obras.



Rua Viúva Cláudio, 291 – Jacaré
Rio de Janeiro – RJ. CEP: 20970-031
Tel: 21 3278-8069/ Fax: 3277-1253
www.altabooks.com.br
e-mail: altabooks@altabooks.com.br

Sumário

O que há de novo no C# 3.0	1
O primeiro programa em C#	4
Compilação	7
Sintaxe	7
Identificadores e palavras-chaves	8
Literais, marcadores de pontuação e operadores	9
Comentários	10
Conceitos básicos de tipos	10
Exemplos de tipos predefinidos	11
Membros de um tipo	12
Conversões	14
Tipos de valor versus Tipos de referência	15
Taxonomia de tipos predefinidos	18
Tipos numéricos	19
Literais numéricos	20
Conversões numéricas	21
Operadores aritméticos	22
Operadores de acréscimo e decréscimo	23
Operações especializadas em integrais	23
Inteiros de 8 e 16 bits	24
Valores especiais de float e double	25
double versus decimal	26
Erros de arredondamento de números reais	26

Tipos e operadores booleanos	27
Operadores de igualdade e comparação	27
Operadores condicionais	28
Strings e Caracteres	29
Conversões char	30
Tipo de string	30
Arrays	32
Inicialização de elemento padrão	33
Arrays multidimensionais	34
Expressões simplificadas de inicialização de array	36
Verificação de limites	37
Variáveis e parâmetros	37
Pilha e Heap	38
Atribuição definitiva	39
Valores padrão	40
Parâmetros	40
var: Variáveis locais tipificadas	
implicitamente (C# 3.0)	44
Expressões e operadores	45
Expressões primárias	46
Expressões void	46
Expressões de atribuição	47
Precedência e associatividade dos operadores	48
Tabela de operadores	49
Instruções	50
Instruções de declaração	50
Instruções de expressão	51
Instruções de seleção	51
Instruções de iteração	55

Instruções de salto	56
Instruções variadas	58
Namespace	58
A diretiva using	60
Regras dentro de um namespace	60
Alias (apelidos) de tipos e namespaces	61
Classes	62
Campos	62
Métodos	62
Construtores de instância	63
Inicializadores de objeto (C# 3.0)	66
A referência this	66
Propriedades	67
Indexadores	69
Constantes	70
Construtores estáticos	71
Classes estáticas	72
Finalizadores	73
Classes e métodos parciais	73
Herança	74
Polimorfismo	75
Casting (declaração)	75
Funções-membro virtuais	77
Classes e membros abstratos	78
Como ocultar membros herdados	78
Como isolar funções e classes	79
A palavra-chave base	79
Construtores e herança	80
Sobrecarga e resolução	81

O tipo object	82
Boxing e Unboxing	83
Verificação estática e dinâmica de tipos	84
Listagem de membros de objeto	84
GetType() e typeof	84
Equals, ReferenceEquals e GetHashCode	85
O método ToString	86
Structs	87
Semânticas de construção da struct	87
Modificadores de acesso	88
Exemplos	89
Limitação da acessibilidade	89
Restrições aos modificadores de acesso	89
Interfaces	90
Extensão de interfaces	91
Implementação explícita de interface	91
Implementação virtual de membros de interface	92
Reimplementação de interface em uma subclasse	93
Enums	94
Conversões de enums	95
Enumerações com Flags	96
Operadores para enums	97
Tipos aninhados	97
Genéricos	98
Tipos genéricos	98
Por que os genéricos existem?	99
Métodos genéricos	100
Declaração de parâmetros genéricos	101
Typeof e os genéricos	102

O valor genérico default	102
Restrições genéricas	103
Genéricos e covariância	104
Subclasses de tipos genéricos	105
A auto-referência das declarações genéricas	106
Dados estáticos	106
Inicialização da coleção genérica	106
Delegates	107
Como criar métodos para plug-ins com delegates	108
Delegates multicast	108
Instância versus Método estático Target	109
Tipos delegate genéricos	109
Compatibilidade dos delegates	109
Eventos	112
Modelo padrão de eventos	113
Acessores de evento	116
Modificadores de evento	117
Expressões Lambda (C# 3.0)	117
Especificação explícita de tipos	
de parâmetros Lambda	119
Expressões Lambda genéricas e os delegates Func	119
Variáveis externas	120
Métodos anônimos	121
Instruções try e exceções	122
A cláusula catch	124
O bloco finally	125
Acionamento de exceções	126
Tipos comuns de exceção	128

Enumeração e iteradores	129
Enumeração	129
Iteradores	130
Semântica dos iteradores	132
Combinação das seqüências	133
Tipos Nullable	135
Conceitos básicos de Null	135
Operadores lifted (levantados)	136
bool?	138
Operadores de aglutinação Null	138
Sobrecarga de operadores	138
Funções operadoras	139
Sobrecarga dos operadores de igualdade e de comparação	140
Conversões implícitas e explícitas personalizadas	141
Métodos de extensão (C# 3.0)	142
Encadeamento do método de extensão	143
Ambigüidade e resolução	143
Tipos anônimos (C# 3.0)	144
LINQ (C# 3.0)	145
Princípios básicos do LINQ	146
Execução diferida	150
Operadores de consulta padrão	151
Operadores de consulta encadeada	155
Sintaxe de query	156
A palavra-chave let	159
Continuação das queries	160
Geradores múltiplos	160
Junção	162

Ordenação	164
Agrupamento	165
OfType e Cast	167
Atributos	168
Classes dos atributos	168
Parâmetros nomeados e posicionais	169
Objetivos do atributo	169
Especificação de atributos múltiplos	170
Criação de atributos personalizados	170
Recuperação de atributos em tempo de execução	171
Códigos unsafe e ponteiros	172
Conceitos básicos do ponteiro	172
Código unsafe	172
A instrução fixed	173
Operador Pointer-to-Member	174
Arrays	174
void*	175
Ponteiros para código não-gerenciado	175
Diretivas pré-processadoras	176
Atributos de condição	177
Pragma warning	177
Documentação XML	178
Tags padrões de documentação XML	179
Visão geral do Framework	181
O Framework central	182
Tecnologias de interface de usuário	187
Tecnologias de backend	190
Tecnologias de sistemas distribuídos	192
Índice Remissivo	195

C# é uma linguagem de programação orientada a objetos de uso geral e com segurança de tipos, cujo objetivo é a produtividade do programador. Para este fim, a linguagem equaciona simplicidade, expressividade e desempenho. A linguagem C# é indiferente à plataforma, porém ela foi criada para funcionar melhor com o Microsoft .NET Framework. O C# 3.0 se destina ao .NET Framework 3.5.

O que há de novo no C# 3.0

Os recursos do C# 3.0 baseiam-se nas capacidades do LINQ (Language Integrated Query). O LINQ habilita as queries para serem escritas diretamente dentro de um programa em C#, à semelhança do SQL, e verificadas estaticamente para correção. Queries podem executar tanto localmente quanto remotamente; o .NET Framework oferece APIs habilitadas para o LINQ em todas as coleções locais, bancos de dados remotos e XML.

Os recursos do C# 3.0 incluem:

- Expressões Lambda
- Métodos de extensão
- Variáveis locais declaradas implicitamente
- Queries de compreensão
- Tipos anônimos
- Arrays declarados implicitamente
- Inicializadores de objetos
- Propriedades automáticas
- Métodos parciais
- Árvores de expressão

Expressões Lambda são como funções em miniatura criadas dinamicamente. Elas são uma evolução natural dos métodos anônimos introduzidos no C# 2.0 e, de fato, completamente subordinadas à funcionalidade dos métodos anônimos. Por exemplo:

```
Func<int,int> sqr = x => x * x;  
Console.WriteLine (sqr(3));           // 9
```

O principal caso de uso no C# ocorre com as queries do LINQ, conforme a seguir:

```
string[] names = { "Tom", "Dick", "Harry" };  
  
// Incluir apenas nomes com >= 4 caracteres:  
  
IEnumerable<string> filteredNames =  
    Enumerable.Where (names, n => n.Length >= 4);
```

Métodos de extensão estendem um tipo existente através de novos métodos, sem alterar a definição do tipo. Eles agem com uma cobertura sintática¹, fazendo com que os métodos estáticos se pareçam com os métodos de instância. Como os operadores de query do LINQ são implementados como métodos de extensão, podemos simplificar a nossa query anterior dessa forma:

```
IEnumerable<string> filteredNames =  
    names.Where (n => n.Length >= 4);
```

Variáveis locais tipificadas implicitamente aceitam omitir o tipo de variável em uma instrução de declaração, permitindo que o compilador o deduza. Como o compilador pode determinar o tipo de `filteredNames`, podemos simplificar nossa query mais detalhadamente:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Sintaxe de compreensão de queries fornece sintaxe ao estilo SQL para a criação de queries. A sintaxe de compreensão pode simplificar certos tipos de queries de forma substancial, bem como servir de cobertura sintática para queries ao estilo Lambda. Eis o exemplo anterior na sintaxe de compreensão:

```
var filteredNames = from n in names  
                   where n.Length >= 4  
                   select n;
```

Tipos anônimos são classes simples criadas dinamicamente, e geralmente são usadas no resultado final das queries:

```
var query = from n in names where n.Length >= 4
            select new {
                Name = n,
                Length = n.Length
            };
```

Eis um exemplo mais simples:

```
var dude = new { Name = "Bob", Age = 20 };
```

Arrays declarados implicitamente eliminam a necessidade de declarar o tipo de array ao construir e inicializar um array em apenas uma etapa:

```
var dudes = new[]
{
    new { Name = "Bob", Age = 20 },
    new { Name = "Rob", Age = 30 }
};
```

Inicializadores de objeto simplificam a construção de objetos permitindo que as propriedades sejam definidas em linha após a chamada do construtor. Os inicializadores de objeto funcionam com os tipos anônimos e nomeados. Por exemplo:

```
Bunny b1 = new Bunny {
    Name = "Bo",
    LikesCarrots = true,
};
```

O equivalente em C# 2.0 é:

```
Bunny b2 = new Bunny( );
b2.Name = "Bo";
b2.LikesCarrots = false;
```

Expressões automáticas reduzem o trabalho de criar propriedades que simplesmente obtêm/definem um campo de suporte privado. No exemplo a seguir, o compilador gera automaticamente um campo de suporte privado para X:

```
public class Stock
{
    public decimal X { get; set; }
}
```

Métodos parciais permitem que uma classe parcial auto-gerada forneça ganchos personalizáveis para autoria manual. LINQ para SQL usa métodos parciais de classes geradas que mapeiam tabelas SQL.

Árvores de expressão são códigos em miniatura de DOMs que descrevem expressões lambda. O compilador do C# 3.0 gera árvores de expressão quando uma expressão lambda é atribuída ao tipo especial `Expression<TDelegate>`:

```
Expression<Func<string,bool>> predicate =  
    s => s.Length > 10;
```

As árvores de expressão possibilitam às queries LINQ executarem remotamente (ex.: em um servidor de banco de dados), porque elas podem ser observadas e interpretadas em tempo de execução (ex.: dentro de uma instrução SQL).

O primeiro programa em C#

Eis um programa que multiplica 12x30 e imprime o resultado, 360, para a tela. A barra dupla indica que o resto da linha é um *comentário*.

```
using System;                                // importando namespace  
  
class Test                                    // declaração de classe  
{  
    static void Main( )                      // declaração de método  
    {  
        int x = 12 * 30;                    // instrução 1  
        Console.WriteLine (x);              // instrução 2  
    }                                        // fim do método  
}                                            // fim da classe
```

No núcleo deste programa residem duas *instruções*. As instruções em C# executam sequencialmente. Cada instrução é finalizada com um ponto-e-vírgula:

```
int x = 12 * 30;  
Console.WriteLine (x);
```

A primeira instrução calcula a *expressão* 12 * 30 e armazena o resultado em uma *variável local*, chamada x, que é um tipo de inteiro. A segunda instrução chama o método `WriteLine` da classe `Console` para imprimir a variável x em uma janela de texto na tela.

Um *método* desempenha uma ação em uma sequência de instruções chamada *bloco de instruções* – um par de chaves que contém zero ou mais instruções. Definimos um método simples chamado Main:

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

Criar funções de alto nível que requisitam funções de nível inferior simplifica um programa. Podemos transformar nosso programa com um método reutilizável que multiplica um inteiro por 12, conforme abaixo:

```
using System;

class Test
{
    static void Main( )
    {
        Console.WriteLine (FeetToInches (30));    // 360
        Console.WriteLine (FeetToInches (100));   // 1200
    }
    static int FeetToInches (int feet)
    {
        int inches = feet * 12;
        return inches;
    }
}
```

Um método pode receber dados de *entrada* da chamada (*caller*) que especifica os *parâmetros*, e pode retornar dados de *saída* para a chamada especificando um *tipo de retorno*. Definimos um método chamado FeetToInches que possui um parâmetro de entrada de dados em pés e um tipo de retorno de saída em polegadas:

```
static int InchesToFeet (int feet) {...}
```

Os *literais* 30 e 100 são os argumentos (parâmetros) passados para o método FeetToInches. O método Main em nosso exemplo possui os parênteses vazios, porque não há parâmetros, e ele é void porque não retorna nenhum valor a quem chama:

```
static void Main()
```

C# reconhece um método chamado `Main` como sinalização do ponto de entrada padrão de execução. O método `Main` pode retornar opcionalmente um inteiro (em vez de `void`) para retornar um valor ao ambiente de execução. O método `Main` também pode opcionalmente aceitar um array de argumentos de `string` (que será preenchido com qualquer argumento passado para o executável). Por exemplo:

```
static int Main (string[] args) {...}
```

NOTA

Um array (como `string[]`) representa um número fixo de elementos de um determinado tipo (consulte a próxima seção “Arrays”).

Métodos são um dos diversos tipos de funções em C#. Um outro tipo de função que usamos foi o *operador* *, usado para executar uma multiplicação. Existem também os *construtores*, as *propriedades*, os *eventos*, os *indexadores* e os *finalizadores*.

Em nosso exemplo, os dois métodos são agrupados em uma classe. Uma *classe* agrupa membros de função e membros de dados para formar um elemento orientado a objetos. A classe `Console` agrupa membros que manipulam a funcionalidade da linha de comando input/output, como o método `WriteLine`. A nossa classe `Test` agrupa dois métodos – o método `Main` e o método `FeetToInches`. Uma classe é uma espécie de *tipo*, que examinaremos mais adiante na seção “Conceitos básicos de tipos”.

No nível externo mais extremo de um programa, os tipos são organizados em *namespaces*. A diretiva `using` disponibiliza o namespace `System` para outros aplicativos, portanto poderíamos tomar como referência `System.Console` sem o prefixo `System`. Poderíamos definir todas as nossas classes dentro do namespace `TestPrograms`, conforme abaixo:

```
using System;

namespace TestPrograms
{
    class Test {...}
    class Test2 {...}
}
```

O .NET Framework é organizado em namespaces aninhados. Por exemplo, esse é o namespace que contém tipos para manipulação de texto:

```
using System.Text;
```

A diretiva `using` encontra-se aí para facilitar; você também pode se referir a um tipo por meio de seu nome inteiramente qualificado, que é o nome do tipo como prefixo de seu namespace, como em `System.Text.StringBuilder`.

Compilação

O compilador do C# compila o código fonte, especificado como um conjunto de arquivos com extensão `.cs`, em um *assembly*. Um *assembly* é a unidade de empacotamento e distribuição no .NET, e ele pode ser um aplicativo ou uma biblioteca. Um console normal ou *aplicativo* Windows possui um método `Main` e é um `.exe`. Uma *biblioteca* é uma `.dll` e é equivalente a um `.exe` sem um ponto de entrada. Seu objetivo é ser requisitada (referenciada) por um aplicativo ou por outras bibliotecas. O .NET Framework é um conjunto de bibliotecas.

O nome do compilador do C# é `csc.exe`. Você também pode usar um IDE como o Visual Studio .NET para chamar `csc` automaticamente, ou compilar manualmente a partir da linha de comando. Para compilar manualmente, salve primeiro o programa em um arquivo como *MyFirstProgram.cs* e, em seguida, invoque `csc` (localizado em `<windows>/Microsoft.NET/Framework`) a partir da linha de comando, conforme abaixo:

```
csc MyFirstProgram.cs
```

Isso gera um aplicativo chamado *MyFirstProgram.exe*. Para gerar uma biblioteca (`.dll`), você digitaria:

```
csc /target:library MyFirstProgram.cs
```

Sintaxe

A sintaxe em C# baseia-se na sintaxe do C e C++. Nesta seção, descrevemos os elementos de sintaxe de C# usando o seguinte programa:

```
using System;

class Test
{
    static void Main( )
    {
        int x = 12 * 30;
        Console.WriteLine (x);
    }
}
```


Identificadores e palavras-chaves

Identificadores são nomes que os programadores escolhem para suas classes, métodos, variáveis e assim por diante. Eis os identificadores do nosso programa de exemplo na ordem em que aparecem:

```
System Test Main x Console WriteLine
```

Um identificador tem que ser uma palavra completa, composta essencialmente de caracteres Unicode que começam com uma letra ou sublinhado. Os identificadores em C# têm maiúsculas e minúsculas distinguidas pelo sistema. Por convenção, os argumentos, as variáveis locais e os campos privados devem ser em camel case (ex.: myVariable), e todos os outros identificadores devem ser em Pascal case (ex.: MyMethod).

Palavras-chaves são nomes reservados pelo compilador que não podem ser usados como identificadores. Eis as palavras-chaves do nosso programa de exemplo:

```
using class static void int
```

Abaixo, a lista completa das palavras-chaves de C#:

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

Evitando conflitos

Se você realmente deseja usar um identificador que conflita com uma palavra-chave, você pode qualificá-la com um prefixo `@`. Por exemplo:

```
class class {...}    // ilegal
class @class {...}   // legal
```

O símbolo `@` não faz parte do identificador, portanto `@myVariable` é o mesmo que `myVariable`.

Palavras-chaves contextuais

Algumas palavras-chaves são *contextuais*, o que significa que elas também podem ser usadas como identificadores – sem o símbolo `@`. A seguir, as palavras-chaves contextuais:

add	get	let	set
ascending	global	on	value
by	group	orderby	var
descending	in	partial	where
equals	into	remove	yield
from	join	select	

Com palavras-chaves contextuais não surge ambigüidade dentro do contexto em que elas são usadas.

Literais, marcadores de pontuação e operadores

Literais são fragmentos de dados primitivos incorporados estaticamente ao programa. Os literais do nosso programa de exemplo são 12 e 30.

Marcadores de pontuação ajudam a demarcar a estrutura do programa. Eis os marcadores de pontuação do nosso programa de exemplo:

```
; { }
```

O ponto-e-vírgula é usado para finalizar uma instrução, e ele permite que as instruções sejam quebradas em várias linhas:

```
Console.WriteLine
(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

As chaves são usadas para agrupar várias instruções dentro de um bloco de instrução.

Operadores transformam e associam expressões. A maioria dos operadores em C# é indicada por um símbolo, como o operador de multiplicação *. Discutiremos detalhadamente os operadores mais adiante neste livro. Eis os operadores que usamos em nosso programa de exemplo:

```
. ( ) * =
```

O ponto se refere a um membro de alguma coisa. Os parênteses são usados ao se declarar ou chamar um método; parênteses vazios são usados quando o método não aceita argumentos. O sinal de igual é usado para *atribuição* (o sinal duplo de igual, =, é usado para comparação de igualdade).

Comentários

O C# oferece dois tipos diferentes de estilo de documentação de código fonte: comentários de uma única linha e comentários de várias linhas. Um comentário de uma única linha começa com uma barra dupla e continua até o fim da linha. Por exemplo:

```
int x = 3;           // comentário sobre atribuir 3 a x
```

Um comentário de várias linhas começa com /* e termina com */:

```
int x = 3;           /* este é um comentário que  
                      ultrapassa duas linhas */
```

Comentários podem incorporar tags de documentação XML (consulte a próxima seção “Documentação XML”).

Conceitos básicos de tipos

Um *tipo* define o esquema de um valor. Um valor é um local de armazenamento indicado por uma variável ou constante. Uma *variável* representa um *valor* que pode mudar, enquanto que uma *constante* representa uma invariante. Criamos uma variável local chamada x em nosso primeiro programa:

```
static void Main( )  
{  
    int x = 12 * 30;  
    Console.WriteLine (x);  
}
```

Todos os valores em C# são *instâncias* de um tipo específico. O significado de um valor, e o conjunto de valores possíveis que uma variável pode ter, é determinado por seu tipo. O tipo de x é int.

Exemplos de tipos predefinidos

Os tipos predefinidos são tipos especialmente suportados pelo compilador. O tipo `int` é um tipo primitivo predefinido de representação do conjunto de inteiros que cabe em 32 bits de memória, de -2^{31} a $2^{31}-1$. Podemos executar funções, tais como as aritméticas, com instâncias de tipo `int`, a saber:

```
int x = 12 * 30;
```

Um outro tipo de C# predefinido é o tipo `string`. O tipo `string` representa uma seqüência de caracteres, tais como “.NET” ou “http://oreilly.com”. Podemos manipular as strings invocando as funções, conforme abaixo:

```
string message = "Olá mundo";
string upperMessage = message.ToUpper( );
Console.WriteLine (upperMessage);           // OLÁ MUNDO

int x = 2007;
message = message + x.ToString( );
Console.WriteLine (message);                // Olá mundo2007
```

O tipo primitivo `bool` possui exatamente dois valores prováveis: `true` e `false`. O tipo `bool` geralmente é usado para acessar o fluxo de execução estabelecido por uma instrução `if`. Por exemplo:

```
bool simpleVar = false;
if (simpleVar)
    Console.WriteLine ("Isso não imprimirá");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
    Console.WriteLine ("Isso imprimirá");
```

NOTA

Em C#, os tipos predefinidos (também chamado de tipos embutidos) são reconhecidos por uma palavra-chave em C#. O namespace `System` no .NET Framework contém muitos tipos importantes que C# não predefine (ex.: `DateTime`).

Exemplos de tipos personalizados

Assim como podemos criar funções complexas a partir de funções simples, podemos criar tipos complexos a partir de tipos primitivos. Neste exemplo, definiremos um tipo personalizado chamado `UnitConverter` – uma classe que serve como um esquema para conversões de unidade: