**Namespaces**

Os namespaces ajudam a resolver esse problema criando um contêiner para

itens, como classes. Duas classes com o mesmo nome não serão confundidas se elas

estiverem em namespaces diferentes.

É uma boa prática definir todas as suas classes em namespaces, e o ambiente do

Visual Studio 2013 segue essa recomendação utilizando o nome do seu projeto como

o namespace de nível mais alto. A biblioteca de classes do .NET Framework também

segue essa recomendação: toda classe no .NET Framework está situada em um namespace.

Por exemplo, a classe *Console* reside no namespace *System*. Isso significa que seu

nome completo é, na verdade, *System.Console*.

**Namespaces e assemblies**

Uma diretiva *using* coloca em escopo os itens de um namespace, e você não precisa

qualificar completamente os nomes das classes no seu código. As classes são

compiladas em *assemblies*. Um assembly é um arquivo que tem, em geral, a extensão

de nome de arquivo .dll, embora programas executáveis com a extensão de

nome de arquivo .exe também sejam assemblies.

Um assembly pode conter muitas classes. As classes de biblioteca abrangidas

pela biblioteca de classes do .NET Framework, como *System.Console*, são fornecidas

nos assemblies instalados no seu computador junto com o Visual Studio. Você descobrirá

que a bibl ioteca de classes do .NET Framework contém milhares de classes.

Se todas fossem armazenadas nos mesmos assemblies, estes seriam enormes e difíceis

de manter. (Se a Microsoft atualizasse um único método em uma única classe,

ela teria de distribuir toda a biblioteca de classes a todos os desenvolvedores!)

Por essa razão, a biblioteca de classes do .NET Framework é dividida em alguns

assemblies, agrupados de acordo com a área funcional a que as classes estão

relacionadas. Por exemplo, um assembly “básico” (na verdade, chamado *mscorlib.*

*dll*) contém todas as classes comuns, como *System.Console*, e outros assemblies

contêm classes para manipular bancos de dados, acessar web services, compilar

GUIs e assim por diante. Se quiser utilizar uma classe em um assembly, você deve

adicionar ao seu projeto uma referência a ele. Então, pode adicionar instruções

*using* ao seu código, colocando em escopo os itens do namespace nesse assembly.

Observe que não há necessariamente uma equivalência 1:1 entre um assembly

e um namespace. Um único assembly pode conter classes definidas para

muitos namespaces e um único namespace pode abranger vários assemblies. Por

exemplo, as classes e itens do namespace *System* são, na verdade, implementados

por vários assemblies, incluindo *mscorlib.dll*, *System.dll* e *System.Core.dll*, dentre

outros. Isso parece muito confuso agora, mas você logo irá se acostumar.

Ao utilizar o Visual Studio para criar um aplicativo, o template que você seleciona

inclui automaticamente referências aos assemblies adequados. Por exemplo,

no Solution Explorer do projeto TestHello, expanda a pasta References. Você

verá que um aplicativo de console contém automaticamente referências a assemblies

chamados *Microsoft.CSharp*, *System*, *System.Core*, *System.Data*, *System.Data.*

*DataExtensions*, *System.Xml* e *System.Xml.Linq*. Talvez você fique surpreso ao ver

que *mscorlib.dll* não está nessa lista. Isso acontece porque todos os aplicativos do

.NET Framework devem usar esse assembly, pois ele contém a funcionalidade de

tempo de execução fundamental. A pasta References lista somente os assemblies

opcionais; é possível adicionar ou remover assemblies dessa pasta, conforme for

necessário.

Para acrescentar referências para assemblies adicionais em um projeto, clique

com o botão direito do mouse na pasta References e então, no menu de atalho

que aparece, clique em Add Reference – você fará isso nos próximos exercícios.

Você também pode remover um assembly, clicando nele com o botão direito do

mouse na pasta References e, então, clicando em Remove.

Importante No Windows 7 e no Windows 8, O Visual Studio 2013 fornece

dois templates para compilar aplicativos gráficos: o template Windows Forms

Application e o template WPF Application. Windows Forms é uma tecnologia

que surgiu no .NET Framework versão 1.0. O WPF, ou Windows Presentation

Foundation, é uma tecnologia aprimorada que apareceu na versão 3.0 do .NET

Framework. O WPF oferece muitos recursos adicionais em relação ao Windows

Forms, e você deve considerar o seu uso no lugar do Windows Forms para todos

os novos desenvolvimentos para Windows 7.

**CAPÍTULO 2**

Variáveis, operadores

e expressões

**Instruções**

*Instrução* é um comando que executa uma ação, como calcular um valor e armazenar

o resultado, ou exibir uma mensagem para o usuário. Você combina instruções para

criar métodos. Para aprender mais sobre métodos, consulte o Capítulo 3, “Como escrever

métodos e aplicar o escopo”, mas, por enquanto, considere um método como uma

sequência nomeada de instruções. *Main*, que foi apresentado no capítulo anterior, é

um exemplo de método.

As instruções em C# seguem um conjunto bem definido de regras que descrevem

seu formato e sua construção. Estas são conhecidas coletivamente como *sintaxe*.

(Por outro lado, a especificação *do que* as instruções fazem é conhecida coletivamente

como *semântica*.) Uma das regras de sintaxe mais simples e mais importantes do C#

diz que você deve terminar todas as instruções com um ponto e vírgula.

**Identificadores**

*Identificadores* são os nomes utilizados para distinguir os elementos nos seus programas,

como namespaces, classes, métodos e variáveis. (Discutiremos as variáveis em breve.)

No C#, você deve seguir as regras de sintaxe abaixo ao escolher os identificadores:

j Você pode utilizar apenas letras (maiúsculas ou minúsculas), dígitos e o caractere

de sublinhado.

j Um identificador deve iniciar com uma letra (ou um sublinhado).

Por exemplo, *resultado*, *\_placar*, *timeDeFutebol* e *plano9* são identificadores válidos,

enquanto *resultado%*, *timeDeFutebol$* e *9plano* não são.

**Variáveis**

*Variável* é um local de armazenamento que contém um valor. Você pode considerar

uma variável como uma caixa na memória do computador que contém informações

temporárias. Você deve atribuir a cada variável em um programa um nome não ambíguo

que a identifique de forma única no contexto em que é utilizada. Um nome de

variável é utilizado para referenciar o valor que ela armazena.

Nomeie variáveis

Adote uma convenção de nomes que torne claras as variáveis definidas. Isso é especialmente

importante se você faz parte de uma equipe de projeto com vários desenvolvedores

trabalhando em diferentes partes de um aplicativo; uma convenção de

nomes consistente ajuda a evitar confusão e pode reduzir a extensão de erros. A lista a

seguir contém algumas recomendações gerais:

Não inicie um identificador com um sublinhado. Embora isso seja válido em C#,

pode limitar a interoperabilidade de seu código com aplicativos compilados em

outras linguagens, como Microsoft Visual Basic.

j Não crie identificadores cuja única diferença seja entre maiúsculas e minúsculas.

Por exemplo, não crie uma variável chamada *minhaVariavel* e outra chamada

*MinhaVariavel* para serem utilizadas ao mesmo tempo, porque será muito fácil

confundi-las. Além disso, a definição de identificadores cuja única diferença seja

a distinção entre maiúsculas e minúsculas pode limitar a reutilização das classes

nos aplicativos desenvolvidos com outras linguagens que não diferem maiúsculas

e minúsculas, como o Visual Basic.

j Comece o nome com uma letra minúscula.

j Em um identificador com várias palavras, comece a segunda palavra e as palavras

subsequentes com uma letra maiúscula. Isso é chamado de notação *camelo*

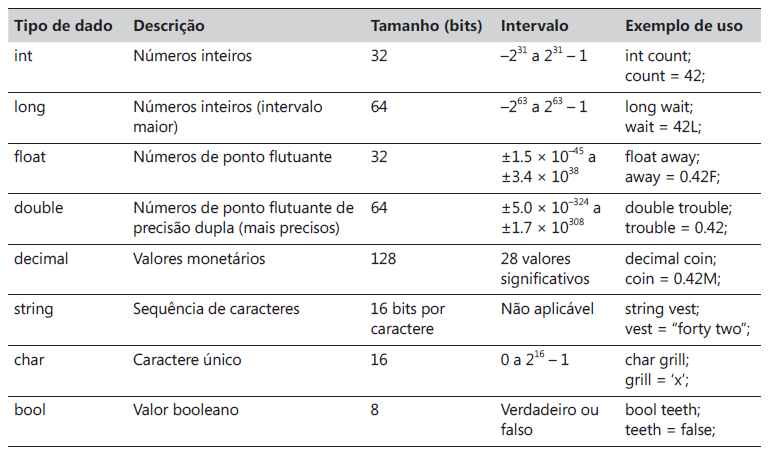
ou *camelCase*.

j Não utilize notação húngara. (Se você for desenvolvedor de Microsoft Visual

C++, provavelmente já conhece a notação húngara. Se não souber o que é isso,

não se preocupe!)

Declare variáveis



Variáveis locais não atribuídas

Quando você declara uma variável, ela contém um valor aleatório até que lhe seja

atribuído um valor. Esse comportamento era uma grande fonte de erros nos programas

C e C++ que criavam uma variável e a utilizavam acidentalmente como fonte de

informações antes de ela receber um valor. O C# não permite utilizar uma variável não

atribuída. É necessário atribuir um valor a uma variável antes de usá-la; caso contrário,

o programa não compilará. Essa exigência é chamada *regra de atribuição definitiva*.

Por exemplo, as instruções a seguir geram a mensagem de erro de tempo de compilação

“Use of unassigned local variable ‘age’” porque a instrução *Console.WriteLine* tenta

exibir o valor de uma variável não inicializada:

int age;

Console.WriteLine(age); // compile-time error

private void showFloatValue()

{

float floatVar;

floatVar = 0.42F;

value.Text = floatVar.ToString();

}

Importante O *F* é um tipo de sufixo especificando que 0.42 deve ser tratado

como um valor *float*. Se você esquecer o *F*, o valor 0.42 será tratado como um

*double* e seu programa não compilará, porque um valor de um tipo não pode

ser atribuído a uma variável de outro tipo sem se escrever código adicional – C#

é muito rígido nesse aspecto.

**Operadores aritméticos**

O C# suporta as operações aritméticas que você aprendeu na escola: o sinal de mais

(+) para adição, o sinal de menos (–) para subtração, o asterisco (\*) para multiplicação

e a barra (/) para divisão. Os símbolos +, –, \* e / são denominados *operadores*

porque “operam” em valores para criar novos valores. No exemplo abaixo, a variável

*moneyPaidToConsultant* termina armazenando o produto de 750 (a diária) e de 20 (o

número de dias que o consultor trabalhou):

long moneyPaidToConsultant;

moneyPaidToConsultant = 750 \* 20;

Nota Os valores nos quais um operador efetua sua função chamam-se *operandos*.

Na expressão 750 \* 20, o \* é o operador e 750 e 20 são os operandos.

**Operadores e tipos**

Nem todos os operadores são aplicáveis a todos os tipos de dados. Aqueles que podem

ser utilizados em um valor dependem do tipo do valor. Por exemplo, você pode

usar todos os operadores aritméticos em valores de tipo *char, int, long, float, double* ou

*decimal*. Contudo, com exceção do operador de adição, *+*, os operadores aritméticos

não podem ser usados em valores de tipo *string* e nenhum deles pode ser usado com

valores de tipo *bool*. Portanto, a instrução a seguir não é permitida porque o tipo *string*

não suporta o operador de subtração (não há sentido em subtrair uma string de outra):

Console.WriteLine("Gillingham" - "Forest Green Rovers");

Contudo, você pode utilizar o operador + para concatenar valores de string.

É preciso ter bastante cuidado, pois isso pode produzir resultados inesperados. Por

exemplo, a seguinte instrução escreve “431” (e não “44”) no console:

Console.WriteLine("43" + "1");

Tipos numéricos e valores infinitos

Há uma ou duas outras características dos números em C# que você precisa

conhecer. Por exemplo, o resultado da divisão de qualquer número por zero é

infinito, estando fora do intervalo dos tipos *int*, *long* e dos tipos *decimais*; consequentemente,

avaliar uma expressão como 5/0 resulta em um erro. Mas os tipos

*double* e *float* têm um valor especial que pode representar valores infinitos, e o

valor da expressão 5.0/0.0 é *Infinity*. A única exceção a essa regra é o valor da

expressão 0.0/0.0. Em geral, se dividir zero por qualquer número, o resultado será

zero, mas se dividir algo por zero o resultado será um número infinito. A expressão

0.0/0.0 resulta em um paradoxo – o valor deve ser zero e infinito ao mesmo

tempo. O C# tem outro valor especial para essa situação, chamado *NaN*, que

significa “not a number” (não é um número). Portanto, se 0.0/0.0 for avaliada, o

resultado será *NaN*.

*NaN* e *Infinity* são propagados pelas expressões. Se 10 + *NaN* for avaliado,

o resultado será *NaN*, e se 10 + *Infinity* for avaliado, o resultado será *Infinity*. A

única exceção a essa regra é quando Infinity é multiplicado por 0. O valor da expressão

*Infinity* \* 0 é 0, embora o valor de *NaN* \* 0 seja *NaN*.

Utilize a associatividade para avaliar expressões

A precedência dos operadores é apenas metade da história. O que acontece quando

uma expressão contém operadores diferentes que têm a mesma precedência? É aí que

a *associatividade* se torna importante. Associatividade é a direção (esquerda ou direita)

em que os operandos de um operador são avaliados. Considere a expressão a seguir

que utiliza os operadores / e \*:

4 / 2 \* 6

À primeira vista, essa expressão é potencialmente ambígua. Qual deve ser efetuada

primeiro, a divisão ou a multiplicação? A precedência dos dois operadores é a

mesma (são ambos multiplicativos), mas a ordem em que são aplicados na expressão

é importante, pois dois resultados diferentes podem ser obtidos:

j Se efetuar primeiro a divisão, o resultado da divisão (4/2) formará o operando

esquerdo do \* operador, e o resultado da expressão inteira será (4/2) \* 6 ou 12.

j Se efetuar primeiro a multiplicação, o resultado da multiplicação (2 \* 6) formará

o operando direito do operador /, e o resultado da expressão inteira será 4 /(2

\* 6) ou 4/12.

Nesse caso, a associatividade dos operadores determina como a expressão é

avaliada. Ambos os operadores, \* e /, associam-se à esquerda, assim, os operandos são

calculados da esquerda para a direita. Nesse caso, 4/2 será avaliado antes da multiplicação

por 6, que resulta em 12.

A associatividade e o operador de atribuição

No C#, o sinal de igual (=) é um operador. Todos os operadores retornam um valor

com base nos seus operandos. O operador de atribuição *=* não é diferente. Ele aceita

dois operandos: o operando à direita é avaliado e então é armazenado no operando

à esquerda. O valor do operador de atribuição é o valor que foi atribuído para o operando

esquerdo. Por exemplo, na seguinte instrução de atribuição, o valor retornado

pelo operador de atribuição é 10, que também é o valor atribuído à variável *myInt*:

int myInt;

myInt = 10; // o valor da expressão de atribuição é 10

Você pode estar pensando que tudo isso é interessante e esotérico, mas e daí?

Bem, como o operador de atribuição retorna um valor, você pode utilizar esse mesmo

valor em outra ocorrência da instrução de atribuição, desta maneira:

int myInt;

int myInt2;

myInt2 = myInt = 10;

O valor atribuído à variável *myInt2* é o valor que foi atribuído a *myInt*. A instrução

de atribuição atribui o mesmo valor a ambas as variáveis. Essa técnica é útil se você

quer inicializar diferentes variáveis com o mesmo valor. Torna-se claro para qualquer

leitor do seu código que todas as variáveis devem ter o mesmo valor:

myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;

A partir dessa discussão, você provavelmente pode deduzir que o operador de

atribuição é associado da direita para a esquerda. A atribuição mais à direita ocorre

primeiro, e o valor atribuído se propaga pelas variáveis da direita para a esquerda. Se

uma das variáveis já tivesse um valor, esse seria sobrescrito pelo valor que está sendo

atribuído.

Entretanto, trate essa construção com cautela. Um erro frequentemente cometido

por novos programadores de C# é tentar combinar esse uso do operador de atribuição

com declarações de variáveis. Por exemplo, você poderia esperar que o código

a seguir criasse e inicializasse três variáveis com o mesmo valor (10):

int myInt, myInt2, myInt3 = 10;

Esse é um código válido do C# (porque é compilado). Ele declara as variáveis

*myInt*, *myInt2* e *myInt3*, e inicializa *myInt3* com o valor 10. Contudo, ele não inicializa

*myInt* nem *myInt2*. Se você tentar utilizar *myInt* ou *myInt2* em uma expressão como

myInt3 = myInt / myInt2;

o compilador gerará os seguintes erros:

Use of unassigned local variable 'myInt'

Use of unassigned local variable 'myInt2'

**Incremente e decremente variáveis**

Se quiser somar 1 a uma variável, pode utilizar o operador +, como demonstrado aqui:

count = count + 1;

Mas adicionar 1 a uma variável é tão comum que o C# fornece um operador

somente para essa finalidade: o operador ++. Para incrementar a variável *count* por 1,

você pode escrever a instrução a seguir:

count++;

Da mesma forma, o C# fornece o operador --, que pode ser utilizado para subtrair

1 de uma variável, desta maneira:

count--;

Os operadores ++ e -- são *unários*, ou seja, eles têm um único operando. Eles

compartilham a mesma precedência e ambos são associativos à esquerda.

Prefixo e sufixo

Os operadores de incremento (++) e decremento (--) fogem do comum, porque você

pode colocá-los antes ou depois da variável. Quando o símbolo do operador é colocado

antes da variável, chamamos de forma *prefixada* do operador, e quando colocado

depois, chamamos de forma *pós-fixada* ou sufixada do operador. Eis alguns exemplos:

count++; // incremento pós-fixado

++count; // incremento prefixado

count--; // decremento pós-fixado

--count; // decremento prefixado

Utilizar a forma prefixada ou sufixada do operador ++ ou -- não faz a menor

diferença para a variável que está sendo incrementada ou decrementada. Por exemplo,

se você escreve *count++*, o valor de *count* aumenta por 1, e se escreve *++count*,

o valor de *count* também aumenta por 1. Sabendo isso, você provavelmente poderia

perguntar por que há duas maneiras de escrever a mesma coisa. Para entender a resposta,

você precisa lembrar que ++ e -- são operadores e que todos os operadores

são utilizados para avaliar uma expressão que tem um valor. O valor retornado por

*count++* é o valor de *count* antes do incremento, enquanto o valor retornado por

*++count* é o valor de *count* depois que o incremento ocorre. Veja um exemplo:

int x;

x = 42;

Console.WriteLine(x++); // x agora é 43, 42 escrito

x = 42;

Console.WriteLine(++x); // x agora é 43, 43 escrito

A maneira de lembrar o que cada operando faz é examinar a ordem dos elementos

(o operando e o operador) em uma expressão prefixada ou sufixada. Na expressão

*x++*, a variável *x* ocorre primeiro; portanto, seu valor é utilizado como o valor da

expressão antes de *x* ser incrementada. Na expressão *++x*, o operador ocorre primeiro;

portanto, sua operação é executada antes de o valor de *x* ser calculado como o

resultado.

Esses operadores são mais utilizados nas instruções *while* e *do*, que serão apresentadas

no Capítulo 5, “Atribuição composta e instruções de iteração”. Caso esteja utilizando

os operadores de incremento e decremento isoladamente, fique com a forma

pós-fixada e seja coerente.

**Declare variáveis locais implicitamente tipadas**

Vimos anteriormente neste capítulo que uma variável é declarada especificando um

tipo de dado e um identificador, assim:

int myInt;

Também foi mencionado que um valor deve ser atribuído a uma variável antes

de se tentar utilizá-la. Você pode declarar e inicializar uma variável na mesma instrução,

como ilustrado a seguir:

int myInt = 99;

Ou assim, supondo que *myOtherInt* seja uma variável do tipo inteiro já inicializada:

int myInt = myOtherInt \* 99;

Agora, lembre-se de que o valor atribuído a uma variável deve ser do mesmo

tipo da variável. Por exemplo, você pode atribuir um valor *int* apenas a uma variável

*int*. O compilador C# pode calcular rapidamente o tipo de uma expressão utilizada

para inicializar uma variável e indicar se não corresponde ao tipo da variável. Também

é possível instruir o compilador C# a deduzir o tipo de uma variável a partir de uma

expressão e utilizá-lo ao declarar a variável com a palavra-chave *var* no lugar do tipo,

como demonstrado aqui:

var myVariable = 99;

var myOtherVariable = "Hello";

As variáveis *myVariable* e *myOtherVariable* são conhecidas como variáveis *implicitamente*

*tipadas*. A palavra-chave *var* faz o compilador deduzir o tipo das variáveis a

partir dos tipos das expressões utilizadas para inicializá-las. Nesses exemplos, *myVariable*

é um *int* e *myOtherVariable* é uma *string*. Contudo, é importante entender que essa

é uma conveniência apenas para declarar variáveis e que, depois que uma variável foi

declarada, você só pode atribuir valores do tipo inferido a ela – valores *float*, *double*

ou *string* não podem ser atribuídos a *myVariable* em um ponto posterior no seu programa,

por exemplo. Você também deve entender que só é possível utilizar a palavra-

-chave *var* quando fornecer uma expressão para inicializar uma variável. A declaração

a seguir é ilegal e causará um erro de compilação:

var yetAnotherVariable; // Erro – o compilador não pode inferir o tipo