Como Programar com ASP.NET e C#

Alfredo Lotar

Novatec

CAPÍTULO 1

Introdução ao C# e .NET Framework

.NET Framework é um componente integrado ao Windows que suporta a execução e o desenvolvimento de uma nova geração de aplicações e XML web services. Segundo a documentação, o .NET Framework foi projetado com os seguintes objetivos:

- Prover um ambiente consistente de programação orientado a objetos de modo que o código do objeto é armazenado e executado localmente, mas pode ser também armazenado na internet e executado remotamente.
- Prover um ambiente de execução de código que minimiza o desenvolvimento de software e conflitos de versão.
- Prover um ambiente de execução de código que promove execução segura de código, inclusive código criado por fontes desconhecidas.
- Prover um ambiente de execução de código que elimine os problemas de desempenho gerados por linguagens de script ou ambientes interpretados.
- Aproveitar o conhecimento do programador em diferentes tipos de aplicações, como aplicações Windows ou web.
- Construir toda a comunicação em padrões reconhecidos pela indústria para que o .NET Framework possa se integrar com qualquer tipo de código.

O .NET Framework tem dois componentes principais: o Common Language Runtime – CLR e o .NET Framework class library, que inclui o ADO.NET, ASP.NET e o Windows Forms.

1.1 Common Language Runtime – CLR

É o mecanismo responsável pela execução das aplicações .NET Framework. O C# suporta CLR, assim como outras linguagens de programação da Microsoft. O código gerado pelo compilador para o suporte CLR chamamos de código gerenciado. O Common Language Runtime – CLR (linguagem comum em tempo de execução) é o

cérebro do .NET Framework. Pense nele como o agente que gerencia o código em tempo de execução, provendo serviços como, por exemplo, o gerenciamento de memória. Veja os benefícios que o CLR nos proporciona:

- Gerenciamento automático de memória.
- Verificação de segurança de tipos.
- Gerenciamento de exceções.
- Segurança aprimorada.
- Acesso a metadados.

1.2 Class library – biblioteca de classes

É uma biblioteca de classes, interfaces e tipos incluídos no .NET Framework que permite acesso às funcionalidades do sistema e é a base a partir da qual são construídas aplicações .NET, componentes e controles. Com essa biblioteca de classes, podemos criar aplicações que executam as mais variadas tarefas como, por exemplo, um software de gestão empresarial, um editor de imagens semelhante ao Photoshop, ou ainda, um web site de comércio eletrônico. As principais funcionalidades oferecidas pela biblioteca de classes são:

- Representa tipos de dados básicos e exceções.
- Permite o encapsulamento da estruturas de dados.
- Executa operações de entrada e saída.
- Acessa informações sobre tipo de dados carregados.
- Realiza verificações de segurança.
- Provê acesso a dados e à internet.
- Permite desenvolver a interface de uma aplicação.
- Permite o desenvolvimento de aplicações Windows e ASP.NET.

1.3 Linguagens suportadas

As linguagens da Microsoft suportadas pelo CLR são: Visual Basic, C#, Visual C++, Jscript, Visual J#, além de linguagens desenvolvidas por outras empresas como, por exemplo, Perl e COBOL.

Uma característica interessante do CLR é a interação entre as linguagens. Por exemplo, podemos desenvolver um componente no Visual Basic e utilizá-lo com C#. Esta é uma característica muito interessante quando trabalhamos com equipes que dominam várias linguagens de programação. Cada programador pode trabalhar usando a sua

linguagem preferida e, no final, o projeto é integrado como se tivesse sido criado em uma única linguagem.

A integração entre as linguagens facilita a vida de empresas e programadores que adquirem ou vendem componentes. A linguagem em que o componente foi desenvolvido é irrelevante. A única preocupação que teremos é que tenha sido desenvolvido numa linguagem que suporte CLR. Pelo que tenho observado em web sites de empresas desenvolvedoras de componentes, C# é a linguagem preferida para desenvolvimento de componentes. Em alguns web sites podemos encontrar a seguinte frase: "Desenvolvido 100% em código gerenciado C#".

1.4 Linguagem intermediária – MSIL

Quando compilamos o código gerenciado, geramos Microsoft Intermediate Language – MSIL, ou simplesmente IL, o qual é independente de CPU e pode ser convertido para código nativo. MSIL inclui instruções para carregar, armazenar, inicializar e executar métodos, assim como instruções para operações aritméticas e lógicas, controle de fluxo etc.

O código contido no MSIL não pode ser executado diretamente; antes de executá-lo, é preciso convertê-lo para instruções que possam ser interpretadas pela CPU. A conversão é realizada por um compilador just-in-time (JIT ou JITter).

MSIL é independente de plataforma, assim só precisamos de um compilador para converter código MSIL em código nativo na máquina-alvo. Além disso, os metadados, que representam informações utilizadas pelo CLR, são colocados em um arquivo chamado Portable Executable – PE, que pode ter a extensão DLL ou EXE.

1.5 Compilando MSIL para código nativo

Antes de executar o MSIL, é preciso utilizar o .NET Framework just-in-time (JIT) para convertê-lo para código nativo. Assim, geramos código específico para a arquitetura na qual roda o compilador JIT. Seguindo este raciocínio, podemos desenvolver uma aplicação e convertê-la para várias plataformas. Para isso, precisamos apenas converter o MSIL para código nativo com um compilador JIT, específico para a plataforma desejada.

Cada sistema operacional pode ter seu compilador JIT. Claro que chamadas específicas a API do Windows não funcionarão em aplicações que estejam rodando em outro sistema operacional. Isso significa que devemos conhecer e testar muito bem uma aplicação, antes de disponibilizá-la para múltiplas plataformas.

Em uma aplicação comercial grande, geralmente usamos um número limitado de funções. Assim sendo, algumas partes do código dessa aplicação podem não ser executadas.

Como a conversão do MSIL para código nativo acarreta consumo de tempo e memória, ela é realizada somente na primeira vez em que o código é executado. Por exemplo, se o nosso programa compila um determinado método, haverá compilação somente na primeira vez em que o método for executado. As chamadas seguintes utilizarão código nativo. O MSIL convertido é usado durante a execução e armazenado para que esteja acessível para chamadas subseqüentes.

Imagine, por exemplo, que você tenha uma classe com cinco métodos; quando você chamar o primeiro método, somente este será compilado, e, quando precisar de outro método, este também será compilado. Chegará um momento em que todo o MSIL estará em código nativo.

1.6 Assemblies

Assemblies são a parte fundamental da programação com .NET Framework. Um assembly contém o código que o CLR executa. O código MSIL dentro de um arquivo portable executable – PE não será executado se não tiver um assembly manifest associado, e cada assembly deve ter somente um ponto de entrada, exemplo: DllMain, WinMain ou Main.

Um assembly pode ser estático ou dinâmico. Assemblies estáticos podem incluir vários tipos (interfaces e classes) do .NET Framework, como também recursos para assemblies (bitmaps, arquivos jpeg etc.). Assemblies estáticos são armazenados no disco rígido com um arquivo portable executable. O .NET Framework cria assemblies dinâmicos que são executados diretamente da memória e não precisam ser armazenados em disco. Podemos salvar em disco assemblies dinâmicos após sua execução.

Um assembly pode ser criado usando o Visual Studio ou outras ferramentas disponibilizadas pelo .NET Framework SDK. Assemblies dinâmicos podem ser criados com as classes da namespace `System.Reflection.Emit`.

1.6.1 Assemblies – benefícios

Assemblies foram projetados para simplificar o desenvolvimento de aplicações e resolver problemas ocorridos pelo conflito de versões causado pela instalação de uma mesma DLL de versão diferente da usada pela aplicação atual. O conflito entre as DLLs é um problema antigo do Windows e não ocorre com o .NET Framework, pois cada aplicação tem suas próprias DLLs. Quando instalamos uma aplicação .NET, os arquivos PE (DLL e EXE) ficam no mesmo diretório da aplicação, assim podemos ter diferentes versões da mesma DLL no computador.

1.6.2 Assembly – conteúdo

Um assembly estático consiste de quatro partes:

- Assembly manifest que contém o metadata do assembly.
- Tipo metadata.
- Código MSIL que implementa os tipos.
- Os recursos para assemblies (bitmaps, arquivos jpeg etc.) e outros arquivos necessários para a aplicação.

1.7 Metadata

O metadata descreve tipos e membros contidos numa aplicação. Quando convertemos o código C# num Portable Executable – PE, o metadata é inserido em uma porção deste arquivo, enquanto o código é convertido para MSIL e inserido em outra porção deste mesmo arquivo. Quando o código é executado, o metadata é carregado na memória, juntamente com as referências para as classes, os membros, a herança etc.

O metadata armazena as seguintes informações:

- Descrição do assembly.
- Identificação (nome, versão, cultura, chave pública).
- Os tipos que são exportados.
- Outros assemblies das quais este assembly depende.
- Permissões de segurança necessárias para a execução.
- Descrição de tipos.
- Nome, visibilidade, classe base e interfaces implementadas.
- Membros (métodos, campos, propriedades, eventos etc.).
- Atributos.
- Elementos descritivos adicionais que modificam tipos e membros.

1.8 Manifest

Todo assembly, estático ou dinâmico, contém uma coleção de dados que descrevem como os elementos em um assembly se relacionam um com os outros. O assembly manifest contém todo o metadata necessário para o assembly definir versão, identificar aspectos relativos à segurança e referências para recursos e classes. O manifest pode ser armazenado junto com MSIL no PE (.dll ou .exe) ou num PE separado. Observe os tipos de assemblies na Figura 1.1:

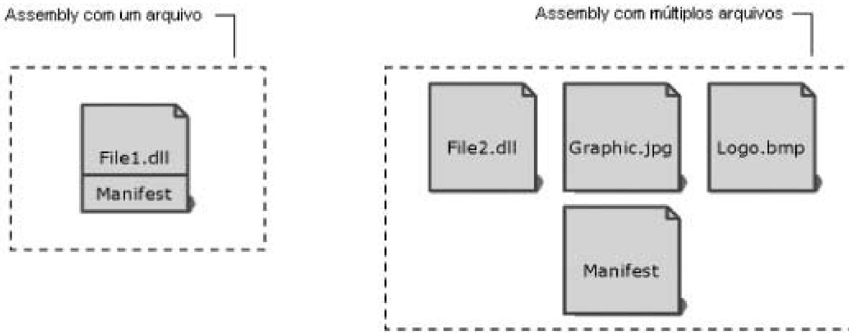

Figura 1.1 – Tipos de assemblies.

1.9 Garbage collector – coletor de lixo

É um mecanismo que descarta, de forma automática, os objetos que não são mais utilizados por uma aplicação. Isso deixa o programador mais tranqüilo, pois não há preocupação com o gerenciamento de memória da aplicação. O CLR detecta quando o programa não está mais usando um objeto e o recicla automaticamente.

1.10 C# – a linguagem de programação

C# (lembrando que se lê C Sharp) é uma linguagem de programação simples, mas poderosa, e ao mesmo, tempo ideal para desenvolver aplicações web com ASP.NET. É uma evolução do C e C++. Além de utilizar muitas características do C++, como, por exemplo, declarações, expressões e operadores, o C# possui um mecanismo chamado Garbage collector (Coletor de Lixo) que gerencia de, forma automática, a memória utilizada pelas aplicações e facilita o desenvolvimento de aplicações web e de aplicações para desktop.

O C# é uma linguagem orientada a objetos com a qual podemos criar classes que podem ser utilizadas por outras linguagens como, por exemplo, o Visual Basic. Uma característica importante é que ainda é possível utilizar os componentes COM, facilitando assim uma rápida migração para um ambiente de desenvolvimento de alto nível sem precisar reescrever todas as aplicações que você possui.

A sintaxe utilizada pelo C# é relativamente fácil, o que diminui o tempo de aprendizado. Depois que você entender como ela funciona, não terá mais motivos para utilizar outra linguagem complicada, pois ela possui o poder do C++ e é simples como o Visual Basic.

Por ser uma linguagem orientada a objeto, existe a capacidade de uma classe herdar certas características ou métodos de outras classes, sejam elas escritas em C# ou em VB.

Todos os programas desenvolvidos devem ser compilados, gerando um arquivo com a extensão DLL ou EXE. Isso torna a execução dos programas mais rápida se compara-

dos com as linguagens de script (VBScript, JavaScript) que atualmente utilizamos na internet.

Nosso primeiro programa C# é extremamente simples. O programa deve exibir na tela “Olá mundo”.

```
// Nosso primeiro programa C#
/* programa Olá mundo
para compilar utilize csc OlaMundo.cs */

namespace OlaMundo {
    class Ola {
        static void Main() {
            System.Console.WriteLine(“Olá Mundo!”);
        }
    }
}
```

Observe o que significa cada linha do programa anterior.

1.10.1 Comentários

Usamos comentários para descrever partes complexas de código, a fim de facilitar a manutenção de quem elaborou o software e de terceiros. Comentários não são interpretados pelo compilador C#. Podemos definir como um comentário: textos, caracteres especiais, trechos de código, números etc.

O C# nos permite definir comentários de duas maneiras: usando barras duplas (`//`) ou os caracteres `/*` e `*/`.

Barras duplas (`//`) convertem o restante da linha em comentários:

```
// Nosso primeiro programa C#
```

Os caracteres `/*` e `*/` definem blocos de texto como comentários. Exemplo:

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

1.10.2 Método Main

Um programa C# deve conter um método `Main`, que controla o início e o fim. Nele você cria objetos e executa outros métodos. Um método `Main` pode não retornar valores.

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

ou retornar um valor inteiro (`int`):

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

Com ambos os tipos (`void` ou `int`), o método `Main` pode conter parâmetros

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

ou

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

1.10.3 Sintaxe C#

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

```
{
    // Código aqui;
}
```

Além disso, C# é sensível a letras maiúsculas e minúsculas, ou seja, `Main` não é o mesmo que `main`. Não observar este detalhe impossibilita a execução do programa.

1.10.4 Entrada e saída

A entrada e saída de um programa C# é realizado pela biblioteca de classes do .NET Framework. A instrução `System.Console.WriteLine("Olá Mundo!");` utiliza o método `WriteLine` da classe `Console`. Não se preocupe se alguns termos são desconhecidos para você, exemplo: classe, método, namespace, diretiva etc. Neste mesmo capítulo, todos esses termos serão explicados e exemplificados.

1.10.5 Compilação e execução do programa

Podemos digitar o código do programa C# no bloco de notas ou utilizar um editor mais completo como, por exemplo, o Microsoft Visual C# Express Edition ou o Visual Studio 2005 ou superior. Para facilitar o aprendizado, utilizaremos o bloco de notas, assim seremos obrigados a realizar todo o trabalho "pesado" para tornar o programa funcional. Para executar um programa desenvolvido em C#, é necessário instalar o .NET Framework. No entanto, antes de instalar no seu computador o .NET Framework 2.0 ou superior, leia o Capítulo 29 deste livro.

Para executar o nosso primeiro programa siga os passos descritos a seguir:

1. Digite o código do programa `OlaMundo` e salve no disco `c:\livro\capitulo1\OlaMundo.cs`. Uma dica: crie um diretório livro, e, em seguida, um subdiretório para cada capítulo do livro. Isso facilitará a localização e a execução dos exemplos deste livro.
2. Em seguida, utilize o Visual Studio Command Prompt: Vá ao menu **Iniciar >> Programas >> Microsoft .NET Framework SDK v2.0 >> SDK Command Prompt**. Veja a Figura 1.2:

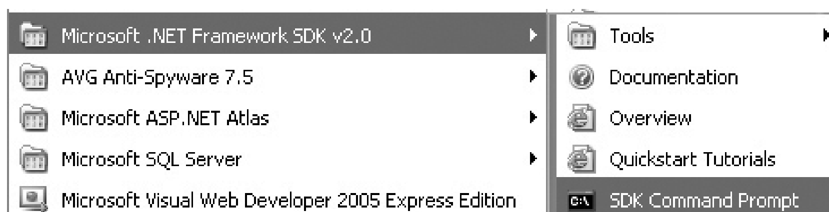

Figura 1.2 – SDK Command Prompt.

3. Mude o diretório atual para `c:\livro\capitulo1`.
4. Digite `csc OlaMundo.cs`.

Neste momento, você está convertendo o código C# em linguagem intermediária (MSIL) e gerando um arquivo Portable Executable – PE do tipo .exe, conforme a Figura 1.3:

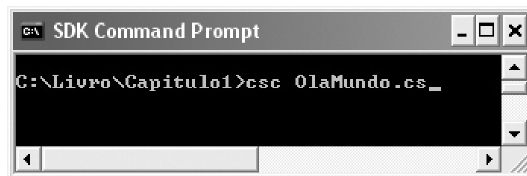

Figura 1.3 – Compilando o programa `OlaMundo`.

Se tudo estiver correto e não houver nenhum erro de sintaxe, aparecerá apenas uma mensagem contendo a versão do compilador C# e informações sobre copyright. Se houver algum erro, será retornada uma mensagem informando a linha onde está o erro, como pode ser visto na Figura 1.4.

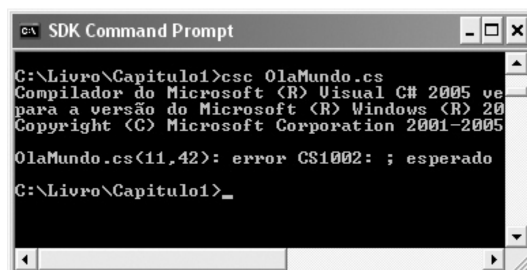

Figura 1.4 – Mensagem de erro do C#.

5. Execute o programa digitando `OlaMundo`. O programa em execução pode ser visto na Figura 1.5.

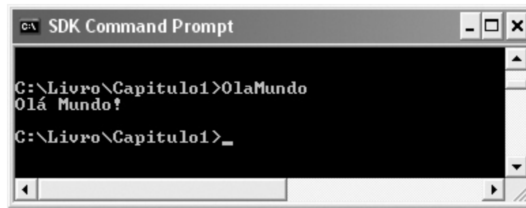

Figura 1.5 – Programa executado com sucesso.

O exemplo de compilação que vimos anteriormente é a forma mais básica que existe, pois não especificamos o local, nem o nome do arquivo executável que estamos gerando, cujo nome pode ser definido como no exemplo que segue:

```
csc /out:c:\saida.exe OlaMundo.cs
```

Se você quiser ver as opções de compilação que podem ser utilizadas em C#, digite apenas `csc /?` na linha de comando, conforme mostra a Figura 1.6:

Figura 1.6 – Parâmetros do comando csc.

Para facilitar a explicação de um bloco ou linha de código, não inserimos junto com o código uma classe e um método `Main`, ou seja, o código não está pronto para execução.

Algumas linhas de código podem aparecer em linha única:

```
Console.WriteLine(i.ToString());
```

Ou em bloco:

```
for (int i = 10; i >= 1; i--) {
   Console.WriteLine(i.ToString());
}
```

Para executar esse código, você precisa criar uma classe e um método `Main` e compilá-lo.

```
class Teste {
    static void Main() {
        for (int i = 10; i >= 1; i--) {
            Console.WriteLine(i.ToString());
        }
    }
}
```

1.10.6 Estrutura de um programa C#

Um programa C# consiste de um ou mais arquivos. Cada arquivo pode conter zero ou mais namespaces. Uma namespace pode conter tipos como classes, `structs`, interfaces, enumerações, e outras namespaces. A seguir, temos um programa simples listado na documentação do .NET Framework que contém todos esses elementos.

```
using System;
namespace NossaNamespace {
    class NossaClass {
    }

    struct NossaStruct {
    }

    interface INossaInterface {
    }

    delegate string NossaDelegate();
    enum NossoEnum {
    }
    namespace OutraNamespace {
        struct OutraStruct {
        }
    }

    class ClasseMain {
        static void Main(string[] args) {
            // Código de inicialização do programa
        }
    }
}
```

1.10.7 Variáveis

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

As variáveis devem conter um nome para que possam ser atribuídos valores a ela. Além do nome, devemos também definir o tipo de dados e o escopo (local onde a variável estará acessível). O escopo é definido pelos modificadores de acesso.

Ao nomear uma variável devemos observar as seguintes restrições:

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

A sintaxe utilizada para declarar uma variável é:

```
Tipo nome;
```

Exemplo:

```
string nome;
int idade=50;
```

Os tipos de dados definem a quantidade de memória que será reservada para a variável.

As variáveis devem ser nomeadas levando-se em conta as tarefas que descrevem, além disso, os nomes devem ser curtos para que possam ser facilmente digitados.

1.10.8 Tipos de dados

Como C# é uma linguagem fortemente tipada (strongly typed). Todas as variáveis e objetos devem ter um tipo declarado.

Os tipos de dados se dividem em `value types` e `reference types`.

Todos os `value types` são derivados de `System.ValueType`, enquanto os `reference types` são derivados de `System.Object`.

Variáveis baseadas em tipos de valor (`value types`) contêm diretamente o valor. Quando copiamos uma variável para outra, uma cópia do valor é passado, enquanto em tipos de referência (`reference types`) somente uma referência do objeto é passada.

Os `values types` se dividem em duas categorias principais: estruturas (`structs`) e enumerações (`enum`).

As estruturas – `struct` se dividem em tipos numéricos (tipos integrais, ponto flutuante e decimal), `bool` e estruturas personalizadas criadas pelos programadores.

A seguir, na Tabela 1.1, temos os tipos integrais com seu tamanho e a faixa de valores que podem conter:

Tabela 1.1 – Tipos integrais

Tipo	Intervalo	Tamanho	Valor padrão
sbyte	-128 até 127	Inteiro de 8 bits com sinal	0
byte	0 até 255	Inteiro de 8 bits sem sinal	0
char	U+0000 até U+ffff	Caractere Unicode de 16 bits	'\0'
short	-32,768 até 32,767	Inteiro de 16 bits com sinal	0
ushort	0 até 65,535	Inteiro de 16 bits sem sinal	0
int	-2,147,483,648 até 2,147,483,647	Inteiro de 32 bits com sinal	0
uint	0 até 4,294,967,295	Inteiro de 32 bits sem sinal	0
long	-9,223,372,036,854,775,808 até 9,223,372,036,854,775,807	Inteiro de 64 bits com sinal	0L
ulong	0 até 18,446,744,073,709,551,615	Inteiro de 64 bits sem sinal	0

Os tipos de ponto flutuante são: `float`, `double`, os quais diferem entre si na faixa e na precisão. como mostra a Tabela 1.2:

Tabela 1.2 – Tipos de ponto flutuante

Tipo	Intervalo aproximado	Precisão	Valor padrão
float	±1.5e-45 até ±3.4e38	7 digitos	0.0F
double	±5.0e-324 até ±1.7e308	15-16 digitos	0.0D

1.10.8.1 Tipo char

Representa um único caractere `Unicode` de 16 bits. É utilizado para representar a maioria das linguagens no mundo. Podemos criar variáveis do tipo `char` e adicionar caracteres:

```
char letra = 'A';
char letra1 = 'H';
```

Uma variável `char` pode conter seqüências de escape hexadecimal (prefixo `\x`) ou uma representação Unicode (prefixo `\u`):

```
char letra2 = '\x0072';   // Hexadecimal
char letra3 = '\u0072';   // Unicode
```

Podemos transformar, de forma explicita, um integral num `char` ou vice-versa.

```
char letra4 = (char)72;   // corresponde a letra H
int numero = (int)'B';    // inteiro 66
```

Combinações de caracteres que consistem de uma barra invertida (\) seguida de uma letra ou combinação de dígitos são chamadas de seqüência de escape. Seqüências de escape são usadas em situações específicas como: salto de linha, retorno de carro,

avanço de página, tabulação horizontal e vertical. Veja a seguir as seqüências de escape usadas no C#:

Seqüência	Significado
\’	Apóstrofo
\”	Aspas
\\	Barra invertida
\a	Alerta
\b	Retrocesso
\f	Avanço de página
\n	Salto de linha
\r	Retorno de carro
\t	Tabulação horizontal
\v	Tabulação vertical

Uma variável precisa ser inicializada antes que possa ser utilizada. Se você declarar uma variável sem declarar um valor inicial, como, por exemplo,

```
int x;
```

será preciso inicializá-la; do contrário não será possível utilizá-la. Podemos inicializar uma variável usando o operador `new`, o qual atribui o valor-padrão do tipo de dados `int` à variável `x`.

```
x = new int();
```

Isso é equivalente a:

```
int x = 0;
```

1.10.8.2 Tipo decimal

O tipo `decimal` é de alta precisão. Ideal para cálculos financeiros e monetários, é um tipo de dados de 128 bits que pode representar valores de aproximadamente

$\pm 1.0 \times 10e{-}28$ até $\pm 7.9 \times 10e28$

com 28 ou 29 dígitos significantes. A precisão é dada em dígitos e não em casas decimais.

O sufixo `m` ou `M` deve ser utilizado para declarar variáveis do tipo `decimal`.

```
decimal x = 102.89m;
```

Sem o sufixo `m` ou `M`, a variável será tratada como sendo do tipo `double`. Um erro é gerado em tempo de compilação.

1.10.8.3 Tipo bool

Representa um valor verdadeiro ou falso. É usado com variáveis ou métodos que retornam o valor `true` ou `false`. O valor-padrão do tipo `bool` é `false`.

```
bool x = true;
bool b = false;
```

Os tipos usados pelo C# podem ser manipulados por `struct` e classes (object e string) do .NET Framework.

Tipo C#	.NET Framework
bool	System.Boolean
byte	System.Byte
sbyte	System.Sbyte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

1.10.8.4 Enumerações – enum

As enumerações nos permitem criar um tipo distinto, constituído de um conjunto de constantes nomeadas. A seguir, temos sua forma mais simples:

```
enum Dias {Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado};
```

Os elementos da enumeração são por padrão do tipo `int`. Os elementos podem ser de qualquer tipo integral, exceto `char`. O primeiro elemento é zero, o segundo elemento é 1, e assim sucessivamente.

Na enumeração anterior Domingo é 0 (zero), Segunda é 1, Terça é 2 etc.

Podemos também atribuir valores arbitrários a cada elemento:

```
enum Dias {Domingo = 10, Segunda = 25, Terça = 48, Quarta = 8, Quinta, Sexta, Sábado};
```

Os elementos aos quais não atribuímos valores arbitrários são incrementados a partir do último elemento com valor.

Domingo é 10, Segunda é 25, Terça é 48, Quarta é 8, Quinta é 9, Sexta é 10, Sábado é 11.

Um tipo integral diferente de `int` pode ser definido:

```
enum Dias:short {Domingo, Segunda, Terça, Quarta, Quinta, Sexta, Sábado};
```

Para acessar o valor de cada elemento, é necessário converter, de forma explícita, cada elemento da enumeração para `int`.

```
int x = (int) Dias.Domingo;
int y = (int) Dias.Sexta;
Console.WriteLine(x);
Console.WriteLine(y);
```

O valor retornado é:

```
0
5
```

Para acessar o valor de vários elementos ao mesmo tempo, temos antes que preparar a enumeração. O atributo `System.FlagsAttribute` deve ser usado na declaração da enumeração:

```
[Flags]
public enum Dias { Segunda = 0x01, Quarta = 0x02};
```

O exemplo completo que segue retorna Segunda, Sexta, Sábado.

```
// Arquivo de exemplo: enum.cs
// Para compilar use: csc enum.cs
using System;
public class FlagEnum {
    [Flags]
    public enum Dias {
        Segunda = 0x01,
        Quarta = 0x02,
        Sexta = 0x04,
        Sábado = 0x08,
    }
    static void Main() {
        Dias d = Dias.Sábado | Dias.Segunda | Dias.Sexta;
        Console.WriteLine(d);
    }
}
```

1.10.8.5 Tipos de referência – reference types

Nas variáveis de tipos de referência (reference types), somente uma referência do objeto é passada. A seguir temos os tipos de referências: class, interface, delegate, object, string e Array.

1.10.8.5.1 Tipo object

Todos os tipos são herdados direta ou indiretamente da classe `Object`. Assim é possível converter qualquer tipo para `object`. O ato de converter um variável value type para `object`

é chamado de `Boxing`. E quando um tipo `object` é convertido para um value type, é chamado de `Unboxing`. O tipo object é representado pela classe `Object` no .NET Framework.

```
object x = 1;
```

1.10.8.5.2 Tipo string

O tipo `string` é o mais utilizado, uma vez que todas as variáveis que não contêm números quase sempre são declaradas como `string`. O tipo `string` é representado por uma `string` de caracteres `Unicode`. Uma string deve estar cercada por aspas duplas `("")`.

```
string b = "texto aqui";
```

O sinal de mais (+) é usado para concatenar uma string:

```
string b = "Concatenando este " + " texto";
string x = "A" + 2;
```

Outro recurso importante é a facilidade de extrair um caractere definido na variável:

```
char extrai = b[5];
```

A linha anterior extrai a letra `t` da variável `b`.

Você pode extrair uma letra de uma palavra:

```
char extrai = "Alfredo"[3];
```

A variável `extrai` retorna a letra r.

O arroba (@) evita que seqüências de escape sejam processadas:

```
@"C:\Livro\Capitulo1\Construtores.cs"
```

É o mesmo que:

```
"C:\\Livro\\Capitulo1\\Construtores.cs"
```

Evita que aspas sejam interpretadas dentro de uma string cercada por aspas duplas.

```
string aspas = @"""abc"" teste";
```

A linha anterior retorna: `"abc" teste`.

1.10.9 Conversões

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

Quando o valor numérico se encontra num controle `TextBox`, ele é uma string e, como você sabe, não podemos realizar cálculos com strings, então a solução é converter essa string para um tipo numérico, como, por exemplo `Int32`, e depois realizar o cálculo. Nem

todas as strings podem ser convertidas para valores numéricos. Veja o caso da string *"Estou escrevendo"*. É impossível convertê-la, pois é uma frase, bem diferente de *"54"*, que se transforma após a conversão no inteiro 54.

Você pode estar se questionando como saberá se pode converter determinado valor para outro sem causar erros. Alguns tipos são convertidos automaticamente se o valor que receber a conversão puder conter todos os valores da expressão. A tabela a seguir nos mostra as conversões que são realizadas automaticamente:

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

No C#, não estamos limitados apenas às conversões automáticas; podemos também forçar uma conversão. Veja a seguir:

```
int a = Int32.Parse(Console.ReadLine());
```

Note que, convertemos o valor inserido no método `ReadLine` para um tipo `int32`.

A seguir, temos as conversões explícitas possíveis:

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

Às vezes, quando fazemos uma conversão explícita, ocorrem certos erros, pois, em situações normais, é impossível converter, por exemplo, um tipo `double` para `byte` sem haver perda de dados.

Podemos controlar se houve ou não algum erro por overflow (valor muito alto) utilizando as instruções `checked` e `unchecked`.

A seguir, veremos dois exemplos: um com a instrução `unchecked` e outro com a instrução `checked`.

```
// Arquivo de exemplo: unchecked.cs
// Para compilar utilize: csc unchecked.cs
using System;
public class unchecked1 {
    public static void Main() {
        int a;
        short b;
        a = 35000;
        unchecked {
            b = (short) a;
            Console.Write(b);
        }
    }
}
```

Nesse exemplo, convertemos um valor (35000) de inteiro para `short`. O valor retornado é:

```
-30536
```

Agora, refaremos o exemplo anterior utilizando a instrução `checked`:

```
// Arquivo de exemplo: checked.cs
// Para compilar utilize: csc checked.cs
using System;
public class checked1 {
    public static void Main() {
        int a;
        short b;
        a = 35000;
        checked {
            b = (short) a;
            Console.Write(b);
        }
    }
}
```

Nenhum valor é retornado, e sim uma mensagem. Uma forma bem mais prática de se fazer conversões é utilizar os métodos da classe `Convert`.

Método	Descrição
`ToBoolean`	Converte uma string para um valor booleano.
`ToByte`	Converte para o tipo `byte`.
`ToChar`	Converte para o tipo `char`.
`ToDateTime`	Converte para o tipo data e hora.
`ToDecimal`	Converte para o tipo `decimal`.
`ToDouble`	Converte para o tipo `double`.

`ToInt16`	Converte para o tipo `short`.
`ToInt32`	Converte para o tipo inteiro.
`ToInt64`	Converte para o tipo `long`.
`ToSingle`	Converte para o tipo `single`.
`ToString`	Converte para o tipo `string`.

Lembre-se de que as conversões realizadas pelos métodos listados na tabela anterior seguem as mesmas regras abordadas anteriormente, ou seja, não podemos converter um tipo `double` para um tipo `int` sem perda de dados.

1.10.10 Operadores

Os operadores são utilizados com expressões. Uma expressão é um fragmento de código que pode ser representado por um valor único, um método, um objeto, um espaço de nome. Uma expressão pode conter um valor literal, invocar um método, operador ou variável.

Um operador é um termo ou símbolo que pode ser usado com uma ou mais expressões, chamadas de operandos. Um operador que possui somente um operando é chamado de operador unário., um operador com dois operandos, por sua vez, é chamado de operador binário.

Operador unário (++)

```
x++;
```

Operador binário (*)

```
x=5 * 8;
```

A tabela a seguir lista os operadores suportados pelo C#. As categorias são listadas em ordem de precedência.

<table>
<tr><th>Categoria</th><th>Operadores</th></tr>
<tr><td>Primários</td><td>x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked</td></tr>
<tr><td>Unários</td><td>+, -, !, ~, ++x, --x, (T)x</td></tr>
<tr><td>Aritméticos</td><td>*, /, %, +, -</td></tr>
<tr><td>Shift</td><td><<, >></td></tr>
<tr><td>Relacional</td><td><, >, <=, >=, is, as</td></tr>
<tr><td>Igualdade</td><td>==, !=</td></tr>
<tr><td>E lógico</td><td>&</td></tr>
<tr><td>OU exclusivo lógico</td><td>^</td></tr>
<tr><td>OU lógico</td><td>|</td></tr>
<tr><td>E condicional</td><td>&&</td></tr>
<tr><td>OU condicional</td><td>||</td></tr>
<tr><td>Condicional</td><td>?:</td></tr>
<tr><td>Atribuição</td><td>=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=</td></tr>
</table>

Você pode criar suas próprias equações, na quais é possível utilizar combinações de operadores. Ao combinar operadores, devemos obedecer a sua precedência, ou seja, devemos obedecer à ordem de processamento dos operadores para realizar os cálculos corretamente. Essa ordem é conhecida como precedência de operadores. Entender como funciona a precedência de operadores é fundamental para se realizar cálculos corretamente. Quando dois operadores com a mesma precedência estão presentes em uma expressão, eles são avaliados da esquerda para direita.

```
int x = 2 + 5 * 8;
```

É atribuído à variável x o total do cálculo: 5 * 8 + 2 é igual 42.

Veja a seguir como você pode obter resultados inesperados se não considerar a ordem de processamento dos operadores:

```
int x= 9 * 3 + 4 / 2;
```

Por exemplo, se você executar o cálculo da esquerda para a direita, neste caso, 9 * 3 é igual a 27; 27 + 4 é igual a 31; 31 / 2 é igual 15,5. Qual seria o resultado correto obedecendo à precedência de operadores?

9 * 3 é igual a 27; 4 / 2 é igual a 2; e 27 + 2 é igual a 29.

Parênteses podem ser usados para cercar parte da expressão e assim forçar para que seja executada antes da outra. Exemplo:

```
int x = 9 * (3 + 5) / 2;
```

No exemplo anterior, os valores contidos nos parênteses são executados primeiro: 3 + 5 é igual a 8; em seguida, é efetuada a multiplicação: 9 * 8 que é igual a 72, e, por fim, a divisão 72 / 2 que é igual a 36.

O resultado armazenado na variável x é 36.

Operações matemáticas que processam cálculos utilizam operadores aritméticos. São imprescindíveis em aplicações financeiras.

```
int v = 2 * 3;    // retorna 6
int d = 6 / 3;    // retorna 2
int r = 13 % 5;   // retorna 3
int s = 5 + 3;    // retorna 8
int m = 5 - 3;    // retorna 2
```

O operador % retorna o resto de uma divisão. 13 % 5 é igual a 3.

Os operadores unários (++x, --x), respectivamente, incrementam e decrementam o valor de uma variável. O ! é um operador lógico de negação. Veja os exemplos a seguir:

```
int x = 1;
++x;   // x agora é igual a 2

int b = 5;
--b;   // b agora é igual a 4
```

```
// Retorna true se o ano de 2006 não for bissexto
if (!DateTime.IsLeapYear(2006)) { }
```

Operadores relacionais comparam valores contidos nos operandos. Os operadores relacionais retornam um valor booleano baseado na comparação realizada.

```
5 < 10
```

Retorna `true`, pois 5 é menor que 10.

```
5 > 10
```

Retorna `false`, pois 5 é menor que 10.

Verificação semelhante pode ser realizada com os operadores menor ou igual (<=) e maior ou igual (>=)

```
5 <= 10
5 >= 10
```

Operador `is` verifica se um objeto é compatível com um determinado tipo.

```
if (obj is string)
{
}
```

E, por último, o operador `as` realiza conversões entre tipos de referência (reference types) compatíveis.

```
object obj = "Alfredo";
string x = obj as string;
```

Operadores de igualdade `==`, `!=` também retornam valores booleanos referentes à comparação realizada entre os operandos.

```
5 == 2
10 != 3
```

Enquanto os operadores de atribuição incrementam ou decrementam os valores dos operandos.

```
int x = 8;    // Atribui 8 a variável x
x += 2;       // Incrementa e armazena o novo valor de x
x -= 3;       // Decrementa e armazena o novo valor de x
x *= 8;       // Multiplica e armazena o valor atual de x. Igual x*8
x /= 2;       // Divide e armazena o valor atual de x. Igual x/2
x %= 3;       // Armazena o resto da divisão em x. Igual x%3
```

Ambos os operandos dos operadores Shift devem ser um tipo integral. O primeiro operando se desloca conforme a quantidade de bits definida no segundo operando. O operador >> faz com que o deslocamento seja para direita, enquanto o operador << realiza o deslocamento para esquerda.

```
int x = 2000;
Console.WriteLine(x >> 4);   // Retorna 125
Console.WriteLine(x << 4);   // Retorna 32000
```

Para entender melhor como funciona o operador `Shift >>`, faremos o cálculo manual até obtermos o valor 125, conforme o exemplo anterior.

O número 4 usado no segundo operando significa que devemos realizar quatro divisões por 2. Exemplo: 2000 / 2 é igual a 1000; 1000 / 2 é igual a 500; 500 / 2 é igual a 250; e, por fim, 250 / 2 é 125. O operador <<, por sua vez, requer uma multiplicação por 2.

1.10.11 Instruções de controle

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

1.10.11.1 Instrução if

A instrução `if` é utilizada para verificar se determinada condição é verdadeira (`true`) ou falsa (`false`). Caso a condição seja verdadeira, os comandos contidos na instrução `if` serão executados. A sintaxe da instrução `if` é a seguinte:

```
if (condição) comandos;
```

De modo que a condição representa qualquer condição lógica de comparação:

- A condição pode ser representada por uma variável.
- Campo de um banco de dados que retorna um valor booleano (`true` ou `false`).
- Expressão que retorna um valor booleano (`true` ou `false`)

O argumento comando representa ação ou tarefa que deve ser realizada caso a condição seja verdadeira.

```
int a = 2;
if (a > 1) Console.WriteLine("Condição verdadeira");
```

Além da instrução `if`, é possível testar a condição de uma expressão com o operador `?:` Sintaxe:

```
condição ? expressão1 : expressão2;
```

Se a condição for verdadeira, a primeira expressão é executada; caso seja falsa, a segunda expressão é executada.

```
int a = 2;
string mensagem = a > 1 ? "Condição verdadeira" : "Condição falsa";
Console.WriteLine(mensagem);
```

Anteriormente, vimos a sintaxe da instrução `if` de linha única, ou seja, os comandos são colocados em uma única linha após a condição. É possível utilizar a instrução `if` em

bloco, ou seja, podemos executar várias linhas de comando se a condição for verdadeira. A sintaxe da instrução `if` em bloco:

```
if (condição) {
    // comando1...
    // comando2...
}
```

Se a condição for verdadeira, todas as linhas de comando serão executadas; mas se a condição for falsa, o programa saltará para a primeira linha após a chave (`}`).

```
int a = 2;
if (a > 1) {
    string mensagem = "Condição verdadeira";
    Console.WriteLine(mensagem);
}
```

É possível aninhar múltiplas instruções `if`.

```
int a = 2;
if (a > 1) {
    if (a == 2) {
        string mensagem = "Condição verdadeira";
        Console.WriteLine(mensagem);
    }
}
```

É claro que podemos executar comandos quando uma condição é falsa. Em determinadas ocasiões, pode ser que você deseje executar comandos quando uma condição for falsa. Há duas formas básicas de se executar comandos quando isso acontecer. A primeira utiliza o operador unário `!`, que inverte a condição real que o segue. Se a condição for verdadeira (`true`), o operador `!` torna toda a expressão falsa (`false`), e vice-versa. A segunda forma utiliza a instrução `else`. Exemplo:

```
if (condição) {
    // Instruções parte verdadeira
}
else {
    // Instruções da parte falsa
}
```

As instruções da parte `else` só serão executadas se a condição contida na instrução `if` não for satisfeita.

```
bool a = true;
if (!a) {
    string mensagem = "Condição verdadeira";
    Console.WriteLine(mensagem);
}
else {
    string mensagem = "Condição falsa";
    Console.WriteLine(mensagem);
}
```

Além das instruções já mencionadas, podemos definir diversas condições dentro de um bloco `if`. Isso pode ser feito com o auxílio da instrução `else if`. A instrução `else if` permite especificar outra condição a ser avaliada se a primeira condição for falsa. É impossível avaliar diversas condições utilizando apenas um bloco `if`. Devemos ressaltar que, independente do número de instruções que forem definidas num bloco `if`, somente os comandos da condição verdadeira serão executados. Exemplo:

```
// Arquivo de exemplo: IfElse.cs
// Para compilar use: csc IfElse.cs

using System;
class TesteIfElse {
    static void Main()    {
        Console.WriteLine("Digite um dos seguintes números: 1, 2, 3, 4");
        int a = Int32.Parse(Console.ReadLine());
        string mensagem = "Variável a igual: ";
        if (a == 1) {
            Console.WriteLine(mensagem + a);
        }
        else if (a == 2) {
            Console.WriteLine(mensagem + a);
        }
        else if (a == 3) {
            Console.WriteLine(mensagem + a);
        }
        else {
            Console.WriteLine(mensagem + a);
        }
    }
}
```

O método `Parse` da estrutura `Int32` converte o valor inserido pelo usuário para um inteiro. Somente valores numéricos são aceitos.

```
int a = Int32.Parse(Console.ReadLine());
```

1.10.11.2 Instrução switch

A instrução `switch` permite a execução condicional de instruções de acordo com os valores de um argumento teste, o qual pode ser uma variável, uma expressão numérica, uma string ou funções. Sintaxe da instrução `switch`:

```
switch (ArgumentoDeTeste) {
    case ExpressãoDoValor:
        // Código a executar, se a condição for verdadeira.
        break;
    default:
        // Código a executar, se nenhuma condição anterior for verdadeira.
        break;
}
```

O argumento de teste definido com `switch` deve ser testado por comparação com os valores das demais instruções `case`. Se o valor do argumento de teste for igual à expressão do valor, então os comandos que seguem a instrução `case` devem ser executados. Se o valor do argumento de teste for diferente da expressão do valor, então o programa compara o valor do argumento de teste com a expressão da próxima instrução `case`.

A instrução `switch` pode incluir qualquer número de instâncias `case`, mas duas declarações `case` não podem ter o mesmo valor. A palavra-chave `break` é obrigatória após cada bloco `case`.

```
// Arquivo de exemplo: switch.cs
// Para compilar use: csc switch.cs
using System;
class Testeswitch {
    static void Main()    {
        Console.WriteLine("Digite um dos seguintes números: 1, 2, 3, 4, 5");
        int a = Int32.Parse(Console.ReadLine());
        string mensagem = "Variável a igual: ";
        switch (a) {
            case 1:
                Console.WriteLine(mensagem + "1");
                break;
            case 2:
                Console.WriteLine(mensagem + "2");
                break;
            case 3:
                goto case 1;
            case 4:
            case 5:
                Console.WriteLine(mensagem + a);
                break;
            default:
                Console.WriteLine(mensagem + a);
                break;
        }
    }
}
```

Às vezes não podemos prever todos os valores possíveis do argumento de teste. Neste caso, utilizamos a instrução `default` dentro do bloco `switch` para realizar uma ação específica caso nenhuma instrução `case` corresponda ao argumento de teste. A instrução `default` é opcional. A palavra-chave `goto` transfere o controle para uma cláusula `case` específica ou para a cláusula `default` e é muito útil em loops do tipo `for`.

1.10.12 Operadores condicionais

Os operadores condicionais permitem testar várias condições no mesmo bloco `if`. Às vezes, queremos executar uma determinada ação somente se determinadas condições forem satisfeitas, por exemplo, se o aluno estiver no terceiro ano do ensino médio e se

tiver mais de 18 anos, então o valor da mensalidade é igual a R$ 500. Como podemos ver, a mensalidade do aluno só é definida quando as condições dos dois argumentos foram satisfeitas. Podemos reescrever o exemplo citado anteriormente usando a instrução `if`.

```
int ano = 3;
int idade = 18;
decimal mensalidade = 350m;
if(ano==3 && idade>18) {
    mensalidade = 500m;
}
```

Como podemos observar, a variável somente é definida com o valor decimal 500 quando as duas condições forem verdadeiras. Na tabela a seguir, temos os principais operadores condicionais utilizados no C#:

Operador	Descrição
&&	Retorna verdadeiro se todos os argumentos forem verdadeiros; retorna falso se pelo menos um argumento for falso.
||	Retorna verdadeiro se pelo menos um argumento for verdadeiro; retorna falso se todos os argumentos forem falso.

Exemplo completo:

```
// Arquivo de exemplo: mensalidade.cs
// Para compilar use: csc mensalidade.cs
using System;
class mensalidade {
    static void Main() {
        Console.WriteLine("Digite um valor maior ou menor que 18:");
        int ano = 3;
        int idade = Int32.Parse(Console.ReadLine());
        decimal mensalidade = 350m;
        if(ano==3 && idade>18) {
            mensalidade = 500m;
        }
        Console.WriteLine("Valor mensal: " + mensalidade);
    }
}
```

1.10.13 Uso de Loops

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

1.10.13.1 Instrução for

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

```
for(tipo Contador = ValorInicial; Contador (< | > | >= | <=) ValorFinal; Contador(++ | --)) {
    // código aqui;
}
```

A variável que serve como contador é definida com o valor inicial, e, a cada vez que o loop é executado, o valor do contador é incrementado (++) ou decrementado (--) e comparado com o valor final. Se o valor do contador for maior que o valor final, o programa salta para a primeira linha após a chave (}).

```
for (int i = 0; i <= 10; i++) {
    // código aqui;
}
```

Podemos sair de loop `for` usando a palavra-chave `goto` quando uma determinada condição for satisfeita. A palavra-chave `goto` faz o programa saltar para a linha onde está a marca. Marca, etiqueta ou label nada mais é do que uma instrução definida pelo programador com o objetivo de redefinir o fluxo do programa.

```
for (int i = 0; i <= 10; i++) {
    if(i == 5) {
        goto FluxoAqui;
    }
}
FluxoAqui:;
```

A palavra-chave `break` também interrompe a execução do loop `for`:

```
for (int i = 0; i <= 10; i++) {
    if(i==5) {
        break;
    }
}
```

A palavra-chave `continue` é muito interessante, pois nos permite saltar para o próximo bloco de execução dentro do loop `for`. Exemplo, considere que o valor atual do contador dentro do loop é um número par. Neste caso, poderíamos utilizar a palavra-chave `continue` para saltar para o próximo bloco. Assim somente os números ímpares seriam impressos na tela.

```
for (int i = 0; i <= 10; i++) {
    if(i % 2 == 0) {
        continue;
    }
    Console.WriteLine(i.ToString());
}
```

Exemplo completo:

```
// Arquivo de exemplo: forteste.cs
// Para compilar use: csc forteste.cs
using System;
class TesteFor {
    static void Main() {
```

```
        for (int i = 0; i <= 10; i++) {
            if(i % 2 == 0) {
                continue;
            }
            Console.WriteLine(i.ToString());
        }
    }
}
```

O exemplo anterior imprime na tela os números: 1, 3, 5, 7, 9. O método `ToString` converte o valor numérico da variável `i` em uma string equivalente.

Podemos colocar um bloco `for` dentro do outro. Assim cada loop `for` superior executa todos os elementos do loop inferior. No exemplo a seguir, o loop `for` com o contador `i` será executado dez vezes, fazendo com que o código no interior do loop seja executado 100 vezes.

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        // código aqui é executado 100 vezes
    }
    // código aqui é executado 10 vezes
}
```

O exemplo a seguir imprime dez vezes o conteúdo da variável `i` e cem vezes o conteúdo da variável `j`.

```
// Arquivo de exemplo: foraninhado.cs
// Para compilar use: csc foraninhado.cs
using System;
class foraninhado {
    static void Main() {
        for (int i = 1; i <= 10; i++) {
            for (int j = 1; j <= 10; j++) {
                Console.WriteLine(j.ToString());
            }
            Console.WriteLine(i.ToString());
        }
    }
}
```

Para realizar uma contagem regressiva, basta definir o contador com um valor negativo. Veja a seguir:

```
for (int i = 10; i >= 1; i--) {
    Console.WriteLine(i.ToString());
}
```

Além de usar o operador `i--`, devemos inverter o valor inicial e o valor final.

1.10.14 Loops condicionais

O C# suporta loops condicionais que nos permitem executar tarefas repetitivas, enquanto determinada condição for satisfeita. A condição retorna um valor booleano (`true`

ou `false`). Assim como acontece com a instrução `if`, a condição pode ser uma função, uma variável booleana ou uma expressão.

1.10.14.1 Instrução while

O primeiro loop condicional que veremos será `while`, que executa os comandos dentro do loop enquanto a condição for verdadeira. Sintaxe da instrução `while`:

```
while(condição) {
    // Código aqui;
    variável (++ | --)
}
```

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

```
int i = 0;
while(i <= 10) {
    Console.WriteLine(i.ToString());
    i++;
}
```

Podemos incrementar a variável `i` ao avaliar a condição:

```
int i = 0;
while(i++ <= 10) {
    Console.WriteLine(i.ToString());
}
```

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

1.10.14.2 Instrução do

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

```
do {
    // Código aqui
    variável (++ | --)
}
while (condição);
```

Exemplo:

```
int x = 0;
do {
   Console.WriteLine(x);
   x++;
}
while (x <= 10);
```

1.10.15 Loop de enumeração

Os loops de enumeração permitem percorrer itens de arrays e coleções. A instrução `foreach` é considerada um loop de enumeração. Cada item de uma coleção é considerado um objeto. A sintaxe da instrução `foreach` é:

```
foreach (tipo elemento in coleção) {
}
```

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

Exemplo com a instrução `foreach`:

```
string[] cores = new string[3];
cores[0] = "Azul";
cores[1] = "Vermelho";
cores[2] = "Verde";
foreach(string cor in cores) {
   Console.WriteLine(cor);
}
```

1.10.16 Classes

As classes nos permitem organizar de forma coerente o código que criamos. Por exemplo, podemos criar uma classe chamada custos que contém todas as funções necessárias para manipular os cálculos referentes aos custos de uma empresa. Quando criamos uma aplicação, podemos criar diversas classes para facilitar a manutenção. No nosso exemplo, a classe `Ola` contém o método `Main`.

Uma classe é o tipo de dados mais poderoso do C#. Classes definem os dados e comportamentos de um tipo de dados.

Os membros de uma classe podem ser: construtores, destrutores, constantes, campos, métodos, propriedades, indexadores, operadores, eventos, delegates, classes, interfaces, `structs`.

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

```
[modificador] class nome {
}
```

Exemplo:

```
public class Taxas {
    // código aqui;
}
```

Para acessar os membros de uma classe, devemos criar uma instância de classe, chamada de objeto, que, por sua vez, é uma entidade concreta baseada em uma classe.

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

Algumas características dos objetos em C#.

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

Os objetos são definidos por classes e `structs` e são criados pela palavra-chave `new`. Para criar uma instância da classe `Taxas`, usamos a seguinte sintaxe:

```
Tipo Referência = new Tipo();
```

Exemplo:

```
Taxas objTax = new Taxas();
```

Quando uma instância de uma classe é criada, uma referência para o objeto é passada para o programador. No exemplo anterior, `objTax` é uma referência para um objeto baseado na classe `Taxas`. A classe define um tipo de objeto, mas não é um objeto.

Podemos criar uma referência para um objeto sem criar um objeto. Exemplo:

```
Taxas objTax;
```

Antes de usar `objTax`, devemos criar o objeto com a palavra-chave `new`, pois o acesso a objetos que não existem falha em tempo de execução.

Acessar os membros de uma classe é muito fácil, basta você criar uma instância da classe com a palavra-chave `new`, e, em seguida, usar a referência para o objeto seguido de um ponto final e o nome do membro. O exemplo a seguir acessa um membro (método) chamado `Calculo`.

```
Taxas objTax = new Taxas();
ObjTax.Calculo();
```

Às vezes, usamos recursos compartilhados que consumem muitos recursos do sistema, por exemplo, o acesso a banco de dados, conexões à internet, acesso a arquivos etc. Nestes casos, devemos liberar os recursos utilizados o mais rápido possível. O bloco `using` é útil em situações em que devemos lidar com esses recursos caros, pois os recursos são imediatamente liberados quando o seu código sai do bloco. Quando não definimos explicitamente que um recurso deve ser liberado, fica a critério do garbage collector liberar o recurso quando não está sendo utilizado pelo programa.

Exemplo:

```
using (Taxas objTax = new Taxas()) {
    // código aqui;
}
```

> **Obs:** somente as classes que implementam a interface `IDisposable` podem ser utilizadas pelo bloco `using`. A interface `IDisposable` possui um método chamado `Dispose` que é utilizado para liberar os recursos do objeto.

Herança de classe e interfaces serão abordadas no segundo capítulo deste livro. A seguir, temos a classe `Taxas` implementando a interface `IDisposable` e o método `Dispose`.

```
public class Taxas : IDisposable {
    void IDisposable.Dispose() {
        // código que libera o recurso aqui;
    }
}
```

1.10.16.1 Modificadores de acesso

Os modificadores de acesso são utilizados para restringir o acesso às classes e a seus membros. Todo o tipo declarado dentro de uma classe sem um modificador de acesso é considerado, por padrão, como `private`. Os modificadores de acesso de classes permitidos são: `public`, `protected`, `internal`, `protected internal`, `private`.

Modificadores	Descrição
`public`	A classe é acessível em qualquer lugar. Não há restrições.
`private`	Acessível apenas dentro do contexto em que foi declarada.
`protected`	A classe é acessível somente na própria classe e nas classes derivadas.
`internal`	A classe é acessível somente no próprio Exe/Dll que a declarou. É o padrão, caso nenhum modificador seja especificado.
`protected internal`	É a união dos modificadores `protected` e `internal`.

Os modificadores de acesso `protected` e `private` só são permitidos em classes aninhadas. Exemplo de classe aninhada.

```
public class Taxas {
    private class Juros {
    }
}
```

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

```
public class Taxas {
    public class Juros {
    }
}
```

Para criar uma instância da classe aninhada, usamos:

```
Taxas.Juros obj = new Taxas.Juros();
```

1.10.17 Classes estáticas – static classes

Classes e membros estáticos podem ser acessados sem a necessidade de se criar uma instância da classe. Uma classe pode ser declarada com o modificador `static`, indicando que só pode conter membros estáticos. Não é possível criar uma instância de uma classe `static` com a palavra-chave `new`.

```
public static class Livro {
    // Membros static aqui
}
```

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

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

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

```
public static class Livro {
    public static string Autor = "Alfredo Lotar";
}
```

Para acessar o campo `Autor`, simplesmente usamos o nome da classe, seguido do ponto final e do nome do campo.

```
Livro.Autor;
```

Exemplo completo:

```
// Arquivo de exemplo: membrosestaticos.cs
// Para compilar use: csc membrosestaticos.cs
using System;
```

```
public static class Livro {
    public static string Autor = "Alfredo Lotar";
}

class TesteLivro {
    static void Main() {
        Console.WriteLine(Livro.Autor);
    }
}
```

Classes estáticas não são muito comuns. A prática mais comum é declarar alguns membros da classe como estáticos e outros não. Assim, não é preciso declarar toda classe como `static`, mas somente os membros que não precisam estar associados a uma instância da classe. Exemplo:

```
public class Livro {
    public static string Autor = "Alfredo Lotar";
    public decimal PrecoUnitario(int frete, int precoNormal) {
        return frete + precoNormal;
    }
}
```

O nome do autor é retornado por um campo `static`, mas o preço é calculado com o método `PrecoUnitario`.

```
Livro lv = new Livro();
Console.WriteLine(lv.PrecoUnitario(12,40));
Console.WriteLine(Livro.Autor);
```

1.10.18 Estruturas – structs

Uma `struct` é semelhante a uma classe, mas é um tipo de valor, enquanto uma classe é um tipo de referência. Uma `struct` é ideal para conter pequenos grupos de variáveis, pois cria objetos leves, conserva memória e nenhuma referência adicional é criada, como acontece com objetos de classe.

Cópias de uma `struct` são criadas e automaticamente destruídas pelo compilador. Assim sendo, um construtor-padrão e um destrutor são desnecessários.

Uma `struct` pode conter construtores, constantes, campos, métodos, propriedades, indexadores, operadores, eventos e tipos aninhados.

Uma `struct` pode conter interfaces, mas não pode ser herdada por outras `structs`.

As estruturas – `struct` possuem a mesma sintaxe de uma classe, embora sejam mais limitadas que as classes.

```
public struct Livro {
    // Código aqui;
}
```

- **Obs:** o modificador `protected` não pode ser usado para declarar `structs`.

Assim como podemos criar uma instância de uma classe, podemos também criar uma instância de uma `struct`, usando a palavra-chave `new`.

```
Livro lv = new Livro("ASP.NET com C#","Alfredo Lotar");
Console.WriteLine("Título: {0}\nAutor: {1}",lv.titulo,lv.autor );
```

Podemos criar de forma direta inicializando as variáveis:

```
Livro lv;
lv.titulo = "ASP.NET com C#";
lv.autor = "Alfredo Lotar";
Console.WriteLine("Título: {0}\nAutor: {1}",lv.titulo,lv.autor );
```

A `struct` utilizada no exemplo anterior é a seguinte:

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

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

As propriedades de uma `struct` são as seguintes:

- Uma `struct` é um tipo de valor, e uma classe é um tipo de referência.
- Podemos criar uma instância de uma `struct` sem usar o operador `new`.
- Não pode conter construtores sem parâmetros.
- Pode implementar interfaces.
- Não permite inicializar instâncias de campos.
- Não pode ser herdada por outras `structs` ou classes e não pode ser a base de uma classe.

Uma instância de uma classe passa uma referência para um método, enquanto uma `struct` passa uma cópia de si mesma.

1.10.19 Métodos

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

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

```
[modificador] tipo nome (Tipo parâmetro)
```

Exemplo:

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

1.10.19.1 Modificadores de acesso de métodos

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

Exemplo: se uma classe for declarada com o modificador `private`, todos os seus membros serão `private`. Uma classe declarada como `protected` pode ter membros `private`, `internal`, `protect` e `protected internal`.

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

Modificadores	Descrição
`public`	O método é acessível em qualquer lugar. Não há restrições.
`private`	Acessível apenas dentro do contexto em que foi declarado, é o padrão, caso nenhum modificador seja especificado.
`protected`	O método é acessível somente na própria classe e nas classes derivadas.
`internal`	O método é acessível somente no próprio Exe/Dll que a declarou.
`protected internal`	É a união dos modificadores `protected` e `internal`.

Programadores com experiência em Visual Basic devem estar procurando um procedimento `Sub` ou `Function`. No C#, métodos cujo valor de retorno é `void` são equivalentes ao procedimento `Sub` do Visual Basic.

Método `Imprimir` no C#.

```
public void Imprimir() {
    // Código aqui;
}
```

Procedimento `Sub Imprimir` no Visual Basic.

```
Public Sub Imprimir()
    // Código aqui
End Sub
```

Métodos que retornam valores diferentes de `void` são, geralmente, chamados de funções.

Método `Calculo` no C#.

```
public int Calculo() {
    return 0;
}
```

Procedimento `Function Calculo` no Visual Basic:

```
Public Function Calculo() As Integer
    Return 0
End Function
```

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

Exemplo:

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

Se o tipo de retorno de um método for diferente de `void`, então o método retorna um valor com a palavra-chave `return`. Uma instrução que contém a palavra-chave `return` retorna um valor para o método que originou a chamada.

A palavra-chave `return` é também utilizada para parar a execução de um método, e, sem ela, a execução é encerrada somente no final do bloco de código.

Métodos que contêm um valor de retorno diferente de `void` devem obrigatoriamente utilizar a palavra-chave `return`.

Exemplo 1 – retorna uma string:

```
public string Mensagem(string str) {
    return str;
}
```

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

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

1.10.19.2 Parâmetros

Os métodos podem conter parâmetros que são utilizados para passar e, em alguns casos, retornar valores de um método. Quando declaramos um método que contém parâmetros, devemos especificar o tipo de dados que o parâmetro pode conter. É aconselhável não declarar variáveis e parâmetros com o tipo `Object`, pois a aplicação deverá arcar com o custo de uma conversão de dados que, geralmente, consome alguns ciclos de CPU.

```
public int Calculo(int x,int y){}
```

- **Obs:** os nomes dos parâmetros devem ser diferentes uns dos outros.

1.10.19.3 Passando parâmetros por valor

É possível passar informações a um método por valor e por referência. A diferença é a seguinte: os valores passados por referência modificam os valores das variáveis passadas para o método, enquanto um parâmetro passado por valor a um método não modifica o valor da variável passada ao método.

A seguir, adaptamos o método `Calculo` da classe `Taxas` para manipular parâmetros por valor. Na verdade, todos os nossos exemplos até o momento passaram informações a métodos por valor.

```
public class Taxas {
    public int Calculo(int x, int y) {
        x += 10;
        return x * y;
    }
}
```

Repare que o parâmetro `x` recebe um incremento de dez itens.

```
Taxas objTax = new Taxas();
int a = 10;
int b = 20;
Console.WriteLine("Método Calculo: " + objTax.Calculo(a, b));
Console.WriteLine("Variável a: " + a);
Console.WriteLine("Variável b: " + b);
```

Como a passagem dos parâmetros foi realizada por valor, as variáveis `a` e `b` não foram alteradas. Veja o resultado do exemplo a seguir:

```
Método Calculo: 400
Variável a: 10
Variável b: 20
```

1.10.19.4 Passando parâmetros por referência

Como já foi dito anteriormente, os valores passados por referência modificam os valores das variáveis passadas para o método. Isso é útil em situações em que os valores

das variáveis passadas para o método devem conter valores atualizados manipulados pelo método.

Um exemplo: uma variável que contém o número de itens de um carrinho de compras. O total de itens pode ser passado a um determinado método e atualizado no momento em que novos itens foram adicionados ao carrinho de compras.

A seguir, temos a classe `Shopping`. Repare que o parâmetro `totalItens` foi declarado com a palavra-chave `ref`. Assim, informamos ao compilador que os valores serão passados ao método por referência, com o parâmetro `totalItens`.

```
public class Shopping {
    public int AddNovoItem(ref int totalItens) {
        totalItens += 1;
        return totalItens;
    }
}
```

Em seguida, criamos uma instância da classe `Shopping` e passamos a variável `xItens` por referência. Novamente, faz-se necessário utilizar a palavra-chave `ref`. Para usar um parâmetro `ref`, ambos, a definição do método e a chamada do método, devem utilizar explicitamente a palavra-chave `ref`.

Obs: os parâmetros passados por referência precisam ser inicializados.

```
Shopping obj = new Shopping();
int xItens = 8;
Console.WriteLine("Método AddNovoItem: " + obj.AddNovoItem(ref xItens));
Console.WriteLine("Variável xItens: " + xItens);
```

O exemplo anterior retorna:

```
Método AddNovoItem: 9
Variável xItens: 9
```

O valor da variável `xItens` foi alterado pelo método `AddNovoItem`.

1.10.19.5 Parâmetros out

Semelhante ao parâmetro `ref` (por referência), os parâmetros `out` podem ser usados para passar um resultado de volta para um método. A variável passada ao método não precisa ser inicializada, como acontece com os parâmetros `ref`.

A seguir, reescrevemos o método `AddNovoItem` da classe `Shopping`. Agora, temos um parâmetro (`totalItens`) passado por valor e outro parâmetro (`saida`) do tipo `out`.

```
public class Shopping {
    public void AddNovoItem(int totalItens, out int saida) {
        totalItens += 1;
        saida = totalItens;
    }
}
```

A chamada do método ocorre da seguinte forma:

```
Shopping obj = new Shopping();
int xOut;
obj.AddNovoItem(8, out xOut);
Console.WriteLine("Variável xOut: " + xOut);
```

Declaramos, além da instância da classe `Shopping`, a variável `xOut` e, em seguida, chamamos o método `AddNovoItem`. Definimos os itens atuais do carrinho e utilizamos o parâmetro `xOut` com a palavra-chave `out`.

O tipo de retorno do método `AddNovoItem` foi definido como `void`, ou seja, não retorna valor; somente a variável `xOut` contém o valor retornado pelo método.

```
Console.WriteLine("Variável xOut: " + xOut);
```

1.10.19.6 Sobrecarga de métodos

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

Podemos declarar numa mesma classe ou `struct` vários métodos com o mesmo nome, mas com parâmetros diferentes. Isso é chamado de sobrecarga de métodos, cuja a principal vantagem é a flexibilidade. Um único método, por exemplo, pode ter várias implementações. Veja a seguir:

```
MessageBox.Show("Isto é um teste");
MessageBox.Show("Olá mundo!", "Título");
MessageBox.Show("Olá","Título",MessageBoxButtons.YesNo);
MessageBox.Show("Último exemplo", "Título", MessageBoxButtons.OK, MessageBoxIcon.Stop);
```

O método `Show` da classe `MessageBox` é implementado com um, dois, três e quatro parâmetros.

Obedeça as seguintes regras ao usar sobrecarga de métodos:

- Todos os métodos sobrecarregados devem ter o mesmo nome.
- Os métodos devem ser diferentes entre si em pelo menos um dos seguintes itens:
  - Número de parâmetros.
  - Ordem dos parâmetros.
  - Tipo de dados dos parâmetros.
  - O tipo de retorno (válido somente em conversões de operadores).

Exemplo:

```
public void teste() {
   System.Console.WriteLine("Método teste()");
}
```

```
public int teste(int i) {
    return i;
}
```

O exemplo completo possui quatro implementações do método `teste`.

```
// Arquivo de exemplo: sobrecarga.cs
// Para compilar use: csc sobrecarga.cs
using System;
public class Livro {
    public void teste() {
        System.Console.WriteLine("Método teste()");
    }
    public int teste(int i) {
        return i;
    }
    public string teste(int a, string str) {
        return str + " " + a;
    }
    public string teste(string str, int a) {
        return a + " " + str;
    }
}
class TesteLivro {
    static void Main() {
        Livro lv = new Livro();
        lv.teste();
        Console.WriteLine(lv.teste(8));
        Console.WriteLine(lv.teste(10,"Segunda string"));
        Console.WriteLine(lv.teste("Primeira string",10));
    }
}
```

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

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

1.10.20 Constantes

Constantes possuem valores conhecidos em tempo de compilação, mas não podem ser modificados, são declaradas como campos e devem ser inicializadas no momento da declaração; são declaradas com a palavra-chave `const`. Sintaxe:

```
[modificador] const tipo nome = valor;
```

Exemplo:

```
class Livro {
    public const int ano = 2007;
}
```

Uma constante deve ser do tipo `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` ou `string`, uma enumeração ou uma referência para um valor nulo (null).

Podemos declarar múltiplas constantes com o mesmo tipo de dados:

```
public class Livro {
    public const int ano = 2010, mes = 09, dia = 12;
}
```

Uma constante utiliza os modificadores de acesso `public`, `private`, `protected`, `internal` ou `protected internal`.

O acesso às constantes é realizado de forma direta (classe.constante), como se fosse um membro estático.

```
int data = Livro.ano;
```

1.10.21 Campos

Campo é uma variável declarada em uma classe, mas fora de qualquer método, que pode conter dados que são passados a uma classe ou `struct`. Sintaxe:

```
[modificador] tipo nome;
```

Exemplo:

```
private string titulo;
private string autor = "Alfredo Lotar";
```

- **Obs:** é recomendável declarar um campo como `private` para que esteja acessível somente no contexto atual. O acesso a campos por classes externas deve ser realizado por intermédio de métodos, propriedades e indexadores.

Um campo pode ser declarado como sendo somente leitura, assemelhando-se a constantes.

```
private readonly string autor = "Alfredo Lotar";
```

1.10.22 Namespaces – espaço de nome

Há alguns anos, os programadores usavam listas contendo funções que poderiam ser utilizadas para desenvolver um programa de computador. Quando um programador precisava de uma função específica ele recorria à lista em que encontrava as funções em ordem alfabética. Com um mínimo de esforço era possível localizar a função desejada.

Essa abordagem, no entanto, já é praticamente impossível, pois o Windows atingiu um nível de complexidade inimaginável e possui milhares de funções. Localizar uma função rapidamente é uma tarefa muito difícil. Se impressas em ordem alfabética, teriam dezenas, senão centenas de páginas.

Para resolver este problema o .NET Framework utiliza espaços de nome. Todas as suas bibliotecas de classes estão organizadas de forma hierárquica na forma de espaços de

nome. A namespace `System` é o principal espaço de nome do .NET Framework, o qual utiliza espaços de nome para organizar uma grande quantidade de classes de forma lógica e organizada, evitando, assim, conflitos de nome de classes.

1.10.22.1 Utilizando espaços de nome

Para utilizar uma determinada classe, devemos antes importar o espaço de nome que contém esta classe. Assim, indicamos em que espaço de nome está a classe ou a função que queremos executar. No C# isso é feito com a instrução `using`. É possível também utilizar os membros de uma classe sem importar o espaço de nome. Neste caso, utilizamos o nome qualificado completo. No nosso primeiro exemplo, OlaMundo, utilizamos o nome qualificado completo do método `WriteLine`.

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

O grande problema de se utilizar um nome qualificado completo é o excessivo esforço de digitação que precisa ser despendido toda vez que você precisa utilizar um método ou propriedade. Quando importamos um espaço de nome, podemos especificar somente o nome do método ou propriedade que pretendemos utilizar. É como no mundo real. As pessoas no dia-a-dia utilizam apenas o primeiro nome para se referir umas às outras e não o nome completo o tempo todo.

No nosso exemplo poderíamos importar a namespace `System`. Veja a seguir como fica o exemplo OlaMundo:

```
// Nosso primeiro programa C#
/* programa Olá mundo
para compilar utilize csc OlaMundo.cs */
using System;
namespace OlaMundo {
    class Ola {
        static void Main() {
            Console.WriteLine("Olá Mundo!");
        }
    }
}
```

Depois de importar o espaço de nome `System`, observamos que a palavra `System` foi omitida.

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

Ao importar um espaço de nome com a instrução `using`, é possível definir um alias (apelido).

```
using Alias = System.Collections;
```

Essa é uma abordagem útil em situações em que devemos especificar o nome qualificado completo. Geralmente, isso acontece quando utilizamos herança de classes. Assim podemos escrever:

```
public class Form1 : Alias.ArrayList {
    // ...
}
```

Em vez de:

```
public class Form1 : System.Collections.ArrayList {
    // ...
}
```

- **Obs:** todas as palavras-chave do C# são escritas em letras minúsculas, exemplo: `using`, `else`, `if`, `do`, `for`, `foreach`, `in`, `while` etc. Não confundir palavras-chave do C# com métodos, classes e propriedades do .NET Framework; por exemplo: `Console` é uma classe, e `WriteLine` é um método.

1.10.22.2 Criando uma namespace

Para criar um espaço de nome utilizamos a instrução `namespace`.

```
namespace ParaTeste {
    // ...
}
```

Um espaço de nome pode conter: outras namespaces, classes, interfaces, `struct`, enumerações e delegate.

Para importar o espaço de nome anterior utilizamos:

```
using ParaTeste;
```

O nome escolhido para um espaço de nome deve indicar a funcionalidade disponível nos membros nele contidos. Exemplo, a namespace `System.Net.Mail` contém tipos que permitem ao programador enviar e-mails.

A documentação do .NET Framework recomenda que os espaços de nome sejam nomeados de acordo com o seguinte critério:

```
<Empresa>.(<Produto>|<Tecnologia>)[.<Característica>][.<Subnamespace>]
```

Utilizar o nome da empresa ou até o seu próprio nome previne que empresas ou desenvolvedores diferentes tenham namespaces com o mesmo nome e prefixo.

Alguns exemplos de namespaces:

```
Microsoft.WindowsMobile.DirectX
Microsoft.Csharp
Microsoft.VisualBasic.FileIO
Alfredo.AdeusSpam
```

No segundo nível, temos o produto ou a tecnologia e, a partir do terceiro nível, a característica e as subnamespaces são opcionais.

1.10.23 Construtores

Sempre que uma classe ou `struct` é criada, um construtor é chamado. Construtores são métodos executados quando um objeto de um determinado tipo é criado. Eles possuem o mesmo nome da classe em que estão contidos e não retornam valor, nem mesmo `void`.

Uma classe ou `struct` pode conter múltiplos construtores com diferentes argumentos. Um construtor sem parâmetros é chamado de construtor-padrão, invocado sempre que uma instância de uma classe é criada com a palavra-chave `new`.

- **Obs:** uma `struct` não pode conter construtores sem parâmetros.

Usando construtores, um programador pode definir valores-padrão ou definir os valores iniciais dos membros. Exemplo:

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

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

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

Podemos reescrever o exemplo com novos construtores, usando sobrecarga de métodos. Exemplo:

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

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

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

Podemos criar uma instância da classe `Livro` usando o construtor-padrão. Neste caso, o valor dos campos `titulo` e `autor` será respectivamente: ASP.NET com C# e Alfredo Lotar.

```
Livro lv = new Livro();
```

Podemos também criar uma instância definindo os valores iniciais dos campos no momento em que uma instância do objeto é criada.

```
Livro lv = new Livro("Livro sobre ASP.NET","Eu mesmo: Alfredo");
```

Os argumentos do construtor `Livro` definem os valores iniciais dos campos.

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

Caso a classe `Livro` não seja declarada com um construtor, o construtor-padrão automaticamente provê valores iniciais para os campos declarados na classe.

```
public class Livro {
    public string titulo;
    public string autor;
    public decimal preco;
}
```

Os valores iniciais de cada tipo foram abordados no início deste capítulo no tópico sobre tipos de dados.

Os campos `titulo` e `autor` são definidos pelo construtor-padrão como nulo (null) e o campo `preco`, como zero.

1.10.23.1 Construtores private

Construtores cujo modificador é `private` são uma categoria especial de construtores. Geralmente, são usados em classes que possuem somente membros estáticos (`static`). É o mesmo que declarar uma classe como `static`.

Não podemos criar uma instância de uma classe que contém somente construtores definidos com o modificador `private`.

Declarar um construtor como `private` evita que o construtor-padrão seja invocado. Exemplo:

```
public class Livro {
    private Livro() { }
    public static string autor = "Alfredo Lotar";
}
```

O campo autor pode ser invocado:

```
Console.WriteLine(Livro.autor);
```

A linha a seguir, gera um erro ao tentarmos compilar o exemplo.

```
Livro lv = new Livro();
```

1.10.23.2 Construtores static

São construtores utilizados para iniciar dados de membros `static` ou executar uma determinada ação uma única vez. Um construtor do tipo `static` é chamado automaticamente antes de a primeira instância da classe ser criada ou antes de qualquer membro estático ser referenciado.

Propriedades de um construtor `static`:

- Não tem nenhum modificador de acesso ou parâmetro.
- Não pode ser invocado diretamente.
- O programador não tem controle do momento em que o construtor `static` será executado dentro do programa.

Exemplo:

```
// Arquivo de exemplo: construtorstatic.cs
// Para compilar use: csc construtorstatic.cs
public class Livro {
    static Livro() {
        System.Console.WriteLine("O construtor static invocado.");
    }
    public static void Paginar() {
        System.Console.WriteLine("O método Paginar invocado.");
    }
}
class TesteLivro {
    static void Main() {
        Livro.Paginar();
    }
}
```

Após a execução, o exemplo retorna:

```
O construtor static invocado.
O método Paginar invocado.
```

1.10.24 Destrutores

Um destrutor é utilizado para destruir instâncias de uma classe e possui as seguintes propriedades:

- Não pode ser definido em `structs`, somente em classes.
- Uma classe pode ter somente um destrutor.
- Destrutores não podem ser herdados nem sobrecarregados.
- São invocados automaticamente.
- Não possuem modificadores ou parâmetros.
- Um destrutor é declarado com o mesmo nome da classe, mas precedido do caractere ~.

Exemplo:

```
public class Livro {
    ~Livro() {
        // Recursos para liberar aqui;
    }
}
```

O exemplo a seguir demonstra o funcionamento de um destrutor. A classe `Autor` implementa, por meio de herança, a classe `Livro`.

```
// Arquivo de exemplo: destrutores.cs
// Para compilar use: csc destrutores.cs
using System;
public class Livro {
    ~Livro() {
        Console.WriteLine("Libera recursos da classe base");
    }
}

public class Autor:Livro {
    ~Autor() {
        Console.WriteLine("Libera recursos da classe derivada Autor");
    }
}

class TesteDestrutor {
    static void Main() {
        Autor aut = new Autor();
    }
}
```

O exemplo retorna:

```
Libera recursos da classe derivada Autor
Libera recursos da classe base
```

O programador não tem controle sobre o destrutor, ou seja, o coletor de lixo do .NET Framework decide o melhor momento para liberar os recursos. Para forçar a liberação imediata dos recursos utilizados, podemos invocar de forma explicita o coletor de lixo. Exemplo:

```
GC.Collect();
```

- **Obs:** devemos ter cuidado especial com a performance da aplicação. Chamadas excessivas ao coletor de lixo têm reflexo negativo na performance da aplicação. Deixe o coletor de lixo cuidar da liberação dos recursos que não estão sendo utilizados pelo programa.

  Recursos caros, que consomem muita memória, ciclos de CPU e largura de banda, como conexões com banco de dados, redes, acesso a arquivos e internet etc., devem liberar os recursos utilizados o mais rápido possível e de forma explícita, usando os métodos `Close` ou `Dispose`.