Conhecendo C# - Lição 1 : Hello C# !!!

por Fabio R. Galuppo

C# (pronunciada “C Sharp”), é uma nova linguagem de programação, da Plataforma .NET, derivada de C/C++, simples, moderna, orientada à objetos e fortemente tipada(type-safe). C# possui o poder do C/C++ aliado a alta produtividade do Visual Basic.

C# será distribuido juntamente com Microsoft Visual Studio 7.0(Visual Studio.NET), e providenciará acesso a toda a plataforma do Next Generation Windows Services(NGWS), que incluem uma poderosa biblioteca de classes e um mecanismo de execução comum.

C# é a linguagem nativa para .NET Common Language Runtime(CLR), mecanismo de execução da plataforma .NET. Isso possibilita a convivência com várias outras linguagens especificadas pela Common Language Subset(CLS). Por exemplo, uma classe base pode ser escrita em C#, derivada em Visual Basic e novamente derivada em C#.

Objetivos da linguagem

- Primeira linguagem “orientada à componentes” da família C/C++:

.NET Common Language Runtime é um ambiente baseado em componentes, e C# é desenhado para facilitar a criação de componentes. Os conceitos de componentes, como propriedades, métodos, eventos e atributos, são fortemente aplicados. Documentação pode ser escrita dentro dos componentes e exportadas para XML.

C# não requer a bibliotecas de tipo(type libraries), arquivos de cabeçalho(header files), arquivos IDL(IDL files). Os componentes criados em C#, são auto-descritivos e não necessitam de processo de registro.

- Tudo é objeto

Em C#, ao contrário de linguagens como Java ou C++, tipos de dados e objetos interagem. C# fornece um “sistema unificado de tipos”, onde todos os tipos são tratados como objetos, sem perda de performance, ao contrário de linguagens como Lisp ou Smalltalk.

- Próxima geração de softwares robustos e duráveis

Software robusto e durável é uma necessidade básica. Muitas aplicações são dificeis de depurar e algumas vezes trazem resultados inesperados em tempo de execução.

Coletor de Lixo(Garbage Collection) que fornece gerenciamento automatico de memória, Excessões(Exceptions) que dispara erros que podem ser tratados, Segurança no tipo de dados (Type-safety) que assegura a manipulação de variáveis e casts e Versão(Versioning), são a recursos encontrados na linguagem para construção dessa categoria de software.

- Preservar seu investimento

C# é montado sobre a “herança” do C++, isso torna confortável a adaptação do programador C++. C# possibilita utilização de ponteiros, na execução de código não gerenciado.

C# permite interoperabilidade com XML, SOAP, componentes COM, DLL e qualquer outra linguagem da Plataforma .NET, matendo integração total com projetos existentes.

Milhões de linhas de código, em C#, são encontradas no .NET, isso permite uma rápida curva de aprendizado e aumento de produtividade.

Next Generation Windows Services(NGWS) SDK

Para utilizar as classes base da plataforma .NET, o ambiente de execução, documentação e o compilador de C#, é necessário instalar o NGWS SDK que pode ser encontrado em http://msdn.microsoft.com/code/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml.

Windows 2000 (incluindo IIS) e Internet Explorer 5.5 são requiridos nesta instalação.

Primeiro programa

Escrevendo o tradicional programa Hello World, em C#:

```
using System;

class Hello{

  public static void Main(){

    Console.WriteLine("Hello World!!!");

  }

}
```

Digite o código acima no seu editor de texto favorito e grave-o com o nome de *Hello.cs*. Para compilar o exemplo acima, no prompt, digite *csc Hello.cs*. Execute o programa digitando *Hello*. Figura 1, mostra compilação e execução da aplicação em C#:

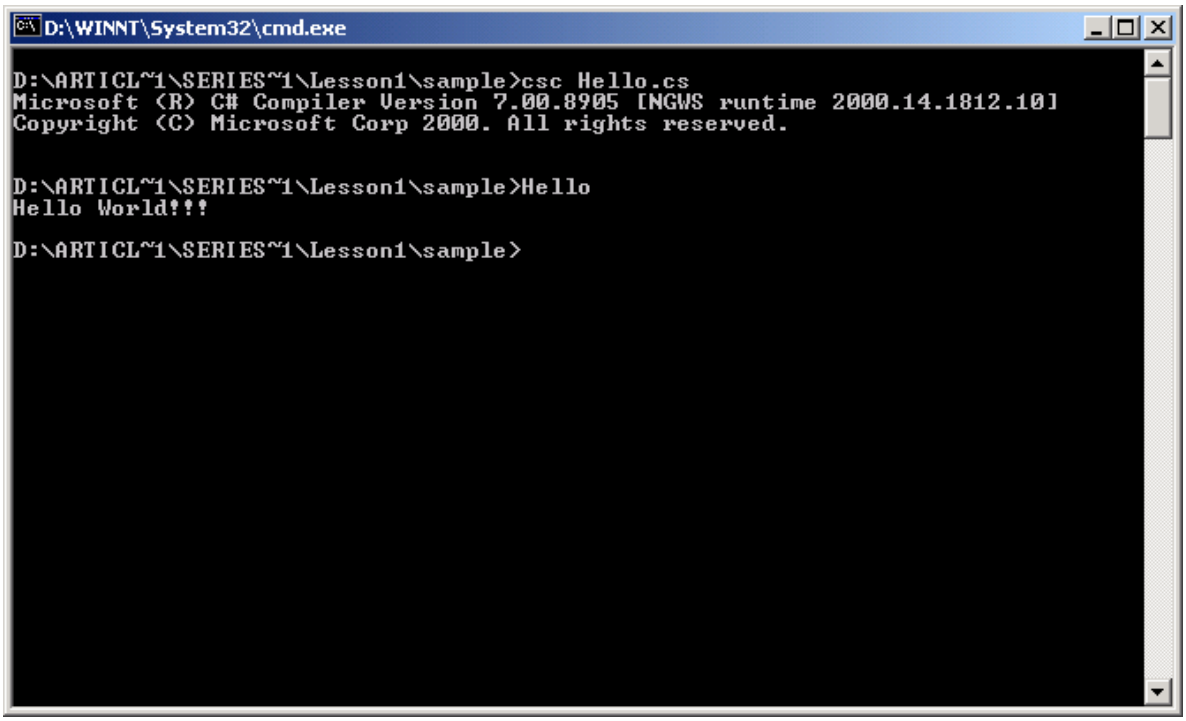

Figura 1: Compilação e Execução do programa em C#

Algumas considerações sobre o código do programa. A claúsula *using* referencia a as classes a serem utilizadas, *System* atua como namespace das classes. O namespace *System* contém muitas classes, uma delas é a *Console*. O método *WriteLine*, simplesmente emite a string no console.

Main

O método *Main* é o ponto de entrada de execução seu programa. No C# não existem funções globais, a classe que será executada inicialmente possui embutida a função estática *Main*. Uma ou mais classe pode conter a função *Main*, portanto apenas uma será o ponto de entrada, indicada na compilação pelo parametro */main:<tipo>*, ou simplificando */m:<tipo>*.

O método *Main*, pode ser declarado de 4 formas:

- Retornando um vazio(void):
  public static void Main()
- Retornando um inteiro(int):
- *public static int Main()*
- Recebendo argumentos, através de um array de string e retornando um vazio:
  public static void Main(string[] args)
- Recebendo argumentos, através de um array de string e retornando um inteiro:
  public static int Main(string[] args)

Estrutura de um programa

O esqueleto de um programa em C#, apresentando alguns dos seus elementos mais comuns, segue abaixo:

```
//Estrutura do programa em C#
using System;

namespace MathNamespace{

  public class MathClass{

    /*
     Main
     Exibe no prompt
    */
    public static void Main(string[] args){

      Math m = new Math();
      Console.WriteLine(m.Sum(1,1));

    }

    //<summary>Classe Math</summary>
    public class Math:Object{

      public int Sum(int a, int b){
        return (a+b);
      }

    }

  }

}
```

A estrutura de um programa em C#, pode ser dividida em um ou mais arquivos, e conter:

- Namespaces;
- Tipos - classes, estruturas, interfaces, delegações, enums;
- Membros – constantes, campos, métodos, propriedades, indexadores, eventos, operadores, construtores;
- Outros - coméntarios, instruções.

Conclusão

Neste artigo, conhecemos qual as caracteristicas da linguagem C# e sua estrutura. Também foi destacado a necessidade do NGWS SDK, que contém o .NET Framework e seus compiladores. Um programa tradicional foi montado, compilado e executado.

Para saber mais. Links:

http://msdn.microsoft.com/library/default.asp?URL=/library/welcome/dsmsdn/deep07202000.htm

http://msdn.microsoft.com/vstudio/nextgen/default.asp
http://www.microsoft.com/net/default.asp
http://msdn.microsoft.com/voices/csharp01182001.asp

Conhecendo C# - Lição 2 : Tipos

por Fabio R. Galuppo

Como toda linguagem de programação o C# apresenta seu grupo de tipos de dados básico. Esses tipos são conhecidos como tipos primitivos ou fundamentais por serem suportados diretamente pelo compilador, e serão utilizados durante a codificação na definição de variáveis, paramêtros, declarações e até mesmo em comparações. A tabela 1 apresenta os tipos básicos(*built-in*) da linguagem C# relacionados juntamente com os tipos de dados do .NET Framework(.NET Types). Em C#, todos eles possuem um correspondente na Common Language Runtime(CLR), por exemplo *int*, em C#, refere-se a *System.Int32*.

Tabela 1: Tipos primitivos do C#

Tipo C#	Tipo .NET	Descrição	Faixa de dados
bool	System.Boolean	Booleano	*true* ou *false*
byte	System.Byte	Inteiro de 8-bit com sinal	-127 a 128
char	System.Char	Caracter Unicode de 16-bit	U+0000 a U+ffff
decimal	System.Decimal	Inteiro de 96-bit com sinal com 28-29 dígitos significativos	$1,0 \times 10^{-28}$ a $7,9 \times 10^{28}$
double	System.Double	Flutuante IEEE 64-bit com 15-16 dígitos significativos	$\pm 5,0 \times 10^{-324}$ a $\pm 1,7 \times 10^{308}$
float	System.Single	Flutuante IEEE 32-bit com 7 dígitos significativos	$\pm 1,5 \times 10^{-45}$ a $\pm 3,4 \times 10^{38}$
int	System.Int32	Inteiro de 32-bit com sinal	-2.147.483.648 a 2.147.483.647
long	System.Int64	Inteiro de 64-bit com sinal	–9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
Object	System.Object	Classe base	
Sbyte	System.Sbyte	Inteiro de 8-bit sem sinal	0 a 255
Short	System.Int16	Inteiro de 16-bit com sinal	-32,768 a 32,767
String	System.String	String de caracteres Unicode	
Uint	System.UInt32	Inteiro de 32-bit sem sinal	0 a 4,294,967,295
Ulong	System.UInt64	Inteiro de 64-bit sem sinal	0 a 18,446,744,073,709,551,615
Ushort	System.UInt16	Inteiro de 16-bit sem sinal	0 a 65,535

Tipos Valor e Tipos Referência

Os tipos de dados no C# são divididos em 3 categorias:

- *Tipos valor(value types)*, que são alocadas na pilha;
- *Tipos referência(reference types)*, que são alocados na *heap*(*Managed Heap*);
- *Tipos ponteiro(pointer types)*, que são ponteiros que poderão ser utilizados em código "inseguro"(unsafe).

Tipos valor armazenam dados em memória enquanto *tipos referência* armazenam uma referência, ou o endereço, para o valor atual. No entanto *tipos referência* não possuem tratamento especial, com relação a operadores, assim como um ponteiro em C++. A figura 1 exibe a alocação de memória dos tipos mais usuais em C#, *tipos valor* e *tipos referência*.

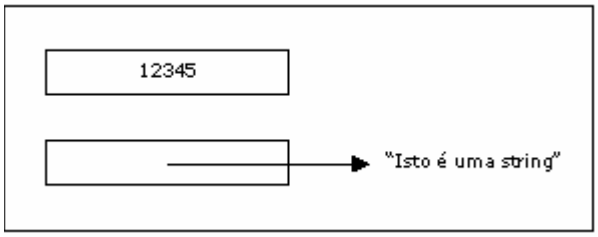

Figura 2: Tipo Valor e Tipo Referência

Quando utilizamos uma variável do *tipo referência* não estaremos acessando seu valor diretamente, mas sim um endereço referente ao seu valor, ao contrário do *tipo valor* que permite o acesso diretamente a seu conteúdo. Diante disso, devemos considerar que a manipulação de variáveis do *tipo valor* oferece uma performance superior a variável do *tipo referência*, por não possuir a necessidade de alocação em *heap*(área de alocação dinâmica) e não ser acessada através de um ponteiro. Normalmente um processo de alocação em *heap* poderá forçar a coleta de lixo(garbage collection), para liberação de memória cosumida. Portanto, utilizar variáveis do *tipo valor*, nos casos que abordaremos mais adiante, é a chave para obter uma performance superior na aplicação.

O código abaixo ilustra a declaração dos tipos previamente discutidos:

```
//Tipo valor
int x = 10;

//Tipo referência
int y = new int(10);
```

No C# os tipos de dados da CLR(.NET Types) também poderão ser utilizados, o código abaixo não possui nenhuma diferença do código anterior, exceto sintática:

```
//Tipo valor
System.Int32 x = 10;

//Tipo referência
System.Int32 y = new System.Int32(10);
```

Podem ser variáveis do *tipo valor*: *bool*, *byte*, *char*, *decimal*, *double*, *enum*, *float*, *int*, *long*, *sbyte*, *short*, *struct*, *uint*, *ulong* e *ushort*. Divididas em duas categorias *estrutura(struct)* e *enumeração(enum)*. A *struct* está classificada em estruturas definidas pelo usuário ou estruturas simples(*built-in*). Considerar os principais casos para o uso de variáveis do *tipo valor*, quando:

- A variável deve atuar como tipo primitivo;
- A variável não necessita herdar de qualquer outro tipo;
- A variável não necessita ser derivada por outro tipo;
- A variável não será passada freqüentemente como parâmetro entre métodos. Isso evitará cópia em mémoria entre as chamadas, que geralmente penalizam a performance.

Ou seja, basicamente quando não há a necessidade de manipular um objeto. Para todos os outros casos aplicam-se a variáveis do *tipo referência*: *class*, *delegate*, *interface*, *object* e *string*. O que deve ser observado no .NET Framework é que uma *string* não é tratada como *tipo referência*, por ser herdada diretamente de *System.Object,* diferentemente de variáveis do *tipo valor* que são herdados de *System.ValueTypes*.

Se for fornecido nenhum contéudo para essas variáveis, por padrão, as do *tipo valor* inicializam com 0, enquanto as do *tipo referência* possuem ponteiro nulo(*null*). O acesso a uma variável não inicializada do *tipo referência* gera uma excessão do tipo *NullReferenceException*.

Conversões Implícitas

Conversões implícitas ocorrem normalmente em atribuições de variáveis e passagem de parâmetros aos métodos. Essas conversões são efetuadas automáticamente quando há necessidade de transformação de dados e estas não forem convertidas explícitamente.

```
//extraindo um valor int de long
int x;
long y = 10;
x = y;
```

A tabela 2 ajudará no suporte há conversões númericas implícitas.

Tabela 2: Conversões Númericas Implícitas suportadas

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

Conversões Explícitas(Casts)

Quando uma variável pode ser mais de 1 tipo, o recurso de *cast* poderá ser utilizado para decidir ou transformar para o tipo desejado.

Cast é uma operação que "extrai" o valor de um determinado tipo para outro, podendo ser resolvido em tempo de compilação ou tempo de execução. O valor obtido é uma representação do valor original, que permanece intacto nessa operação. Este recurso é muito comum em C#, principalmente em técnicas como *boxing* e *unboxing*.

```
//extraindo um valor int de long
int x;
long y = 10;
x =(int)y;
```

```
//extraindo um valor System.Int32 de System.Int64
```

```
System.Int32 x;
System.Int64 y = 10;
x = (System.Int32)y;
```

Nos exemplos acima a operação de *cast* é efetuada nas últimas linhas, onde extrai-se um valor inteiro(*int*) de um inteiro longo(*long*). O que deve ser levado em consideração, tanto para as conversões implícitas quanto para as explícitas, é a faixa de dados, pois o *int* não comporta o grupo *long* totalmente(vide Faixa de Dados na tabela 1), porém o inverso é verdadeiro. Se essa faixa "estourar" o valor obtido não será consistente. Porém este tipo de operação é válida e amplamente utilizada. Por exemplo:

```
class Casts{

  public static void Main(){

    int x;
    long y=10;

    //Cast consistente
    x=(int)y;
    System.Console.WriteLine("Cast consistente int x={0} long y={1}",x,y);

    //Cast inconsistente
    y=2147483648;
    x=(int)y;
    System.Console.WriteLine("Cast inconsistente int x={0} long y={1}",x,y);

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc Casts.cs*. Execute o programa digitando *Casts*. A Figura 2, mostra compilação e execução da aplicação em C#.

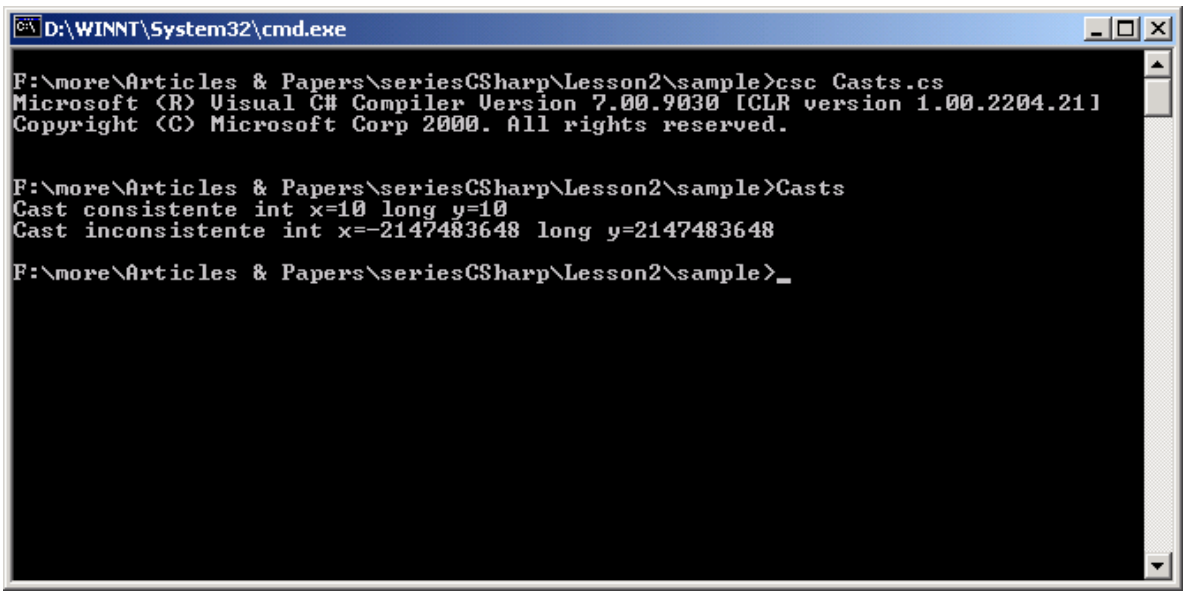

Figura 3: Compilação e Execução do exemplo Casts

A tabela 3 ajudará no suporte há conversões númericas explícitas.

Tabela 3: Conversões Númericas Explícitas suportadas

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

Estrutura(Struct) e Classe(Class)

Em C#, uma estrutura(struct) é sempre tratado como *tipo valor*, enquanto uma classe(class) é sempre tratado como *tipo referência*. Ambos são parecidos, suportam construtores(*constructors*), constantes(*constants*), campos(*fields)*, métodos(*methods*), propriedades(*properties)*, indexadores(*indexers*), operadores(*operators*) e tipos aninhados(*nested types*). Mas as estruturas não têm suporte a recursos relacionados ponteiros, ou melhor referências, tais como membros virtuais, construtores parametrizados, ponteiro *this* e membros abstratos.

```
using System;

//Classe
public class MyClassRect{

  //Campos
  public uint x, y;

  //Calculo da área do retângulo
  public ulong Area(){ return x*y; }

}

//Estrutura
public struct MyStructRect{

  //Campos
  public uint x, y;

  //Calculo da área do retângulo
  public ulong Area(){ return x*y; }

}


class MainClass{

  public static void Main(){

    MyClassRect  cl = new MyClassRect(); //alocado na heap
    MyStructRect st;                     //alocado na stack

    cl.x = 10;
    cl.y = 5;
```

```
    st.x = 10;
    st.y = 5;

    Console.WriteLine("Classe - {0} m X {1} m = {2} m²" ,cl.x,cl.y,cl.Area());
    Console.WriteLine("Estrutura - {0} m X {1} m = {2} m²",st.x,st.y,st.Area());

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc Class_Struct.cs*. Execute o programa digitando *Class_Struct*. A Figura 3, mostra compilação e execução da aplicação em C#.

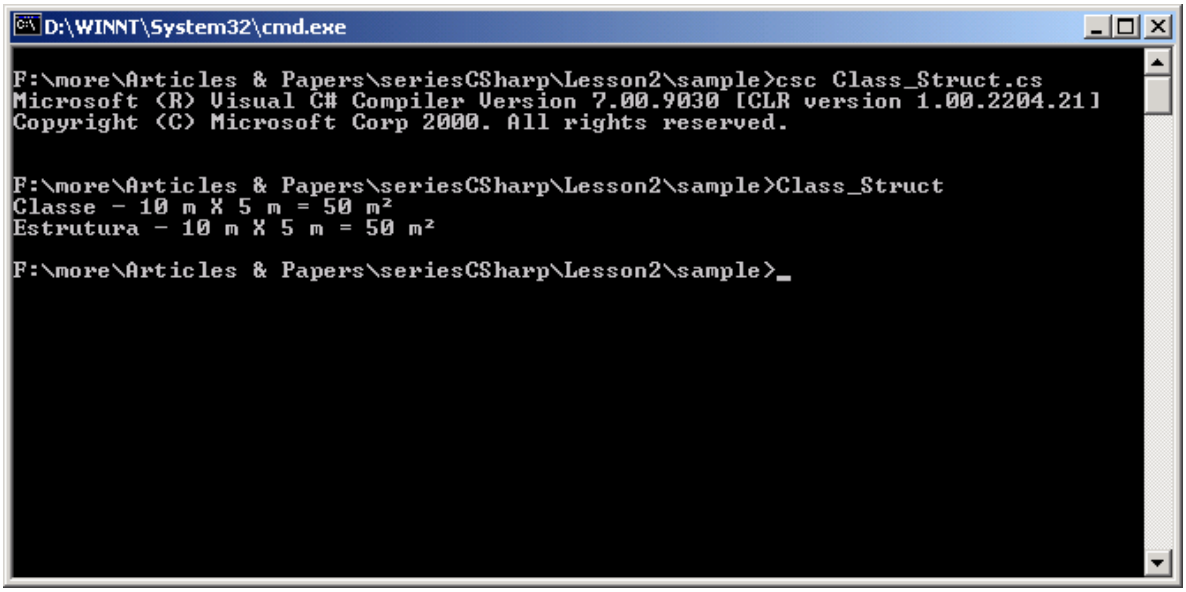

Figura 4: Compilação e Execução do exemplo Class_Struct

Boxing e Unboxing

Diferentemente de algumas linguagens orientada à objetos, o C# não exige um *wrapper*(empacotador) para manipular objetos e tipos, uma técnica muito mais poderosa é utilizada neste caso *boxing* e *unboxing*.

Boxing é a conversão de um *tipo valor* para um *tipo referência* e *unboxing* é o processo contrário desta conversão, ou seja extrai-se o conteúdo do *tipo valor* de um *tipo referência* herdado de *System.ValueType*. Somente conversões explícitas são suportadas para *unboxing*.

```
//Boxing
int x = 12345;
object o = x;

//Unboxing
x = (int)o;
```

Os tipos de dados herdados por *System.ValueTypes* são alocados em *stack*, por serem implementados como estrutura(*struct*), vide documentação de *System.Int32* no .NET Framework SDK, por exemplo. Quando uma variável do *tipo valor* é utilizada, apenas seu contéudo é alocado em memória, no entanto se essa variável sofrer uma operação de *boxing*, seu valor será copiado para a *heap* e será acrescentado o *overhead*(excesso) do objeto *System.ValueTypes*(objeto herdado de *System.Object*, raíz de todos objetos do .NET Framework). A figura 4 exibe o estado do exemplo anterior que manipula um inteiro nas duas formas: *boxing* e *unboxing*.

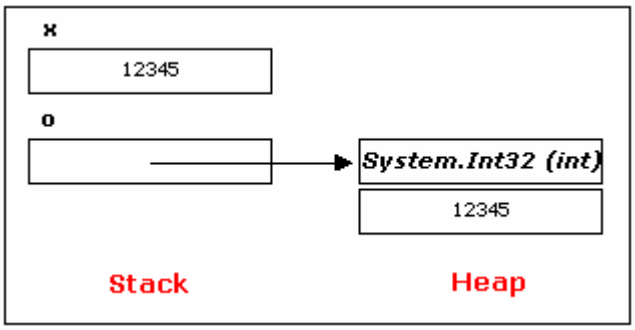

Figura 5: Representação em memória de uma variável int nas formas *boxing* e *unboxing*

O que deve ficar claro no processo de conversão, é que o *boxing* pode ser tratado automáticamente, ou seja implícito, por exemplo para um inteiro(*int*) tornar-se um objeto(*object*) poderemos encontrar *object o = 12345*, neste caso o valor *12345* é tratado como inteiro(*int*). No entanto na forma *unboxing*, o tratamento explícito é requerido *int x = (int)o*, assumindo que *o* representa um *object*. Se a partir da forma *boxing* de um valor, através de conversão explícita, deseja-se obter a forma *unboxing* do mesmo, o *cast* no processo de *unboxing* deverá ser coerente com o tipo específicado, caso contrário um excessão *InvalidCastException* será disparada, isto também é válido para *casts* inconsistenes. O código abaixo demonstra isso:

```
long l = 12345;
object o = (long)l;

//Unboxing inconsistente
int i = (int)o; //excessão InvalidCastException
```

Segue a correção para o exemplo acima:

```
long l = 12345;
object o = (long)l;

//Unboxing consistente
int i = (int)(long)o;
```

Uma outra forma de utilizar as conversões é atráves da operação através de seus membros, ou seja, a partir de um conteúdo ou de uma variável é possível especificar a função ou método de conversão desejado. Se uma variável ou valor é de um tipo determinado, por exemplo, de um tipo *int* é correto utilizar *i.ToInt64()* ou *12345.ToInt64()* para obter-se um valor do tipo *long*. O código anterior pode ser reescrito para:

```
object o = 12345.ToInt64(); //Valor tratado na forma boxing

//Unboxing consistente
int i = (int)(long)o;
```

De acordo com documentação do .NET Framework, se utilizarmos o método *WriteLine* da classe *Console* com uma string formatadora e vários tipos de dados, esses tipos serão tratados como um vetor(*array*) de objetos, conforme figura 5. Então quantos *boxings* serão executados na linha abaixo?

```
long l = 1000;
int i = 500;
Console.WriteLine("{0} {1} {2} {3} {4} {5}",l,i,i,l,l,i);
```

```
public static void WriteLine(
   string format,
   params object[] arg
);
```

Figura 6: Uma das possibilidades do método WriteLine da classe Console

Seriam necessários 6 *boxings* para última linha, um para cada variável a ser tratada como objeto. O processo de *boxing* consome memória e tempo de execução da CPU, portanto o código acima é eficientemente reescrito para:

```
long l = 1000;
int i = 500;
object ol = l;
object oi = i;
Console.WriteLine("{0} {1} {2} {3} {4} {5}",ol,oi,oi,ol,ol,oi);
```

Neste caso o número de *boxings* é reduzido 2. Este tipo de cuidado sempre deverá ser tomado para a obtenção da melhor performance do código, resultando menos tempo de processamento e recursos consumidos.

Conclusão

Neste artigo, é apresentado como o C# trata a declaração, utilização e a conversão de tipos de dados, conhecidos também como tipos primitivos ou fundamentais. Juntamente foi aborda o procedimento de *boxing* e *unboxing*, que é utilizado largamente na construção do código na plataforma .NET. Bem como algumas regras para eficiência na utilização desse recurso.

Links

http://msdn.microsoft.com/msdnmag/issues/1200/dotnet/dotnet1200.asp
http://msdn.microsoft.com/vstudio/nextgen/default.asp

http://www.microsoft.com/net/default.asp
http://msdn.microsoft.com/voices/csharp02152001.asp
http://msdn.microsoft.com/net/
http://gotdotnet.com/

Conhecendo C# - Lição 3 : Comandos

por Fabio R. Galuppo

C# possui um grupo variado de comandos de controle de fluxo do programa que definirão sua lógica de processamento. A forma com que os comandos são tratados permance similar às anteriores. Portanto, na programação sequêncial, na programação estruturada ou programação orientada à objetos, cada comando é executado igualmente, ou seja, instrução por instrução. Na verdade, esta é a lei do processador. Os principais comandos são classificados em:

- Declaração
- Rótulo ou Label
- Desvio ou Jump
- Seleção
- Iteração ou Loop

A maioria destes comandos é familiar para os programadores de linguagens como C ou C++.

Estes comandos podem ser encontrados dentro de um bloco ou na forma isolada. O bloco de comandos é representado por chaves ({}). Normalmente, um bloco de comando é tratado como uma unidade de comando e é utilizado com instruções como *if*, *for*, *while*, *try*, entre outros.

```
//Comando isolado
if(a==true) System.Console.Write("Verdadeiro");
```

```
//Bloco de comandos
{

int x = 100;
int y = 200;
int z = x + y;

}
```

Uma linha de comando em C# pode conter uma ou mais instruções. Todas elas são terminadas pelo finalizador (*end point*) ponto e vírgula (;).

```
//Vários comandos em uma única linha
while(a<=100) if(b==true) a++;
```

Um tipo de comando conhecido como vazio (*empty*) também é válido em C#.

```
//Loop infinito sem comando ou comando vazio
for( ; ; ) ;
```

```
//Loop infinito com comando. Último ponto e vírgula representa o finalizador
for( ; ; )  System.Console.Write("C# é legal");
```

Declaração

Indica um recurso a ser utilizado ou exposto. As declarações podem ser de variáveis, constantes, funções (métodos), propriedades ou campos.

Para declarar uma variável basta especificar o tipo seguido de uma literal. A definição de uma constante ocorre da mesma forma que a variável, porém é prefixada a a palavra-chave *const* e obrigatóriamente sufixada com o sinal de igual (=) e seu valor. A atribuição para uma variável pode ocorrer a qualquer momento dentro da aplicação ou na sua definição inicial, no entanto, para uma constante, somente a atribuição inicial é válida e permanecerá inalteralda durante a execução. Uma variável ou constante é dependente de escopo, ou seja, da visibilidade dentro de uma aplicação. Por exemplo, a variável *x* abaixo não pode ser utilizada diretamente fora da função *Main.* Neste caso, ela deve ser utilizada através de um argumento de uma função ou outra variável com escopo mais elevado. Os principais escopos são local e classe. Todas as variáveis ou constantes dentro do escopo de classe podem, obviamente, ser utilizadas dentro da classe como um todo. Para não alargarmos a discussão, não citarei, neste momento, detalhes de orientação à objetos tais como propriedades, métodos, campos e visibilidades internas e externas (*public*, *private, internal* ou *protected*). O exemplo abaixo ilustra os casos mais usuais de declaração:

```
using System;

class Declares{

  private static int f = 1000, g;        //Variáveis de escopo de classe
  private const int m = 1000, n = 10000; //Constantes de escopo de classe

  public static void Main(){

    //Constantes de escopo local
    const int  x = 10;
    const long y = 100;

    //Variáveis de escopo local
    int  a = 10;
    long b;

    b = 100;
    g = 10000;

    printf(x,y,a,b);

  }

  //Função
  private static void printf(int ix, long ly, int ia, long lb){

    Console.WriteLine("Locais\nconstantes: x={0} y={1} váriaveis: a={2}
b={3}",ix,ly,ia,lb);
    Console.WriteLine("Classe\nconstantes: m={0} n={1} váriaveis: f={2}
g={3}",m,n,f,g);

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc Declares.cs*. Execute o programa digitando *Declares*. A Figura 1, mostra compilação e execução da aplicação em C#.

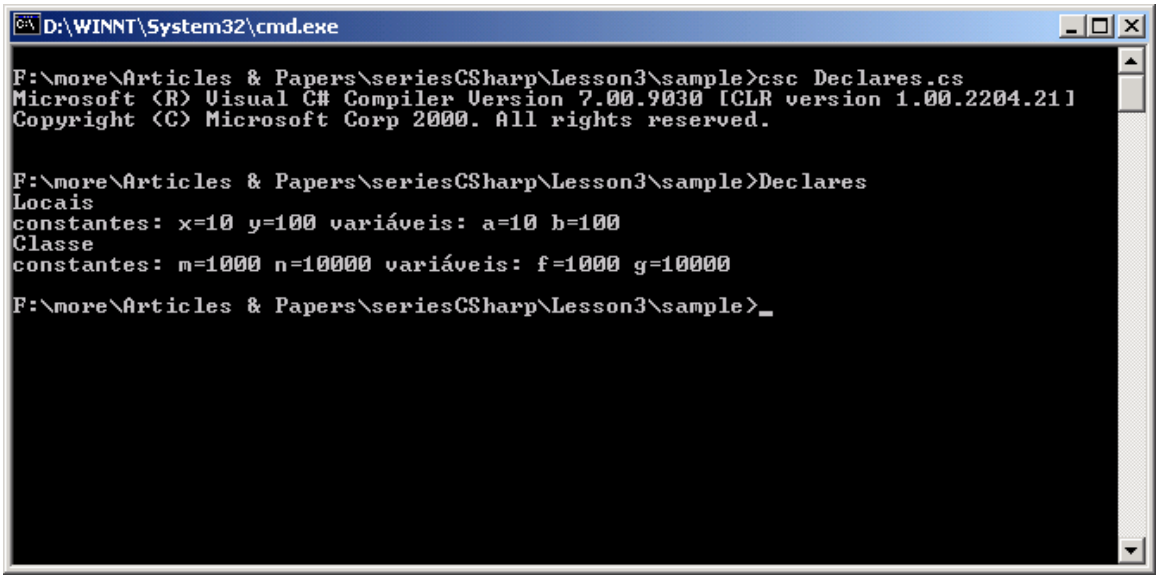

Figura 7: Compilação e Execução do exemplo Declares

As declarações de variáveis ou constantes de mesmo tipo podem ser feitas na mesma linha.

```
int x;
int y;
int z = 10;
```

Pode ser o mesmo que:

```
int x, y, z = 10;
```

Sobre escopo de variáveis e constantes um escopo interno tem prioridade sobre o escopo externo, para qualquer resolução sempre o interno será utilizado. Variáveis ou constantes dentro do mesmo escopo não poderão possuir o mesmo nome, pois o compilador tratará isto como um conflito. Porém, elas poderão ser encontradas com o mesmo nome em escopos ou locais diferentes. O conflito de nomes também deve ser considerado para funções (métodos), propriedades e campos.

```
class MyClass{

  private int x = 10;

  private int MyFunction(){
    int x = 100;
    System.Console.Write(x); //Exibe o valor 100
  }
  private int MyFunction2(){ System.Console.Write(x); //Exibe o valor 10 }

}
```

Rótulo (Label)

Discutir sobre rótulos (labels) sem falar sobre comandos de desvio é impossível. Mas a única função de um rótulo é marcar, com uma palavra não reservada, uma área do código que pode ser saltada através do comando *goto*. Mesmo que C# permitida, este tipo de prática não é aconselhável em linguagens de alto ou médio nível desde a evolução da programação sequencial.

```
using System;

class Jumps{

  public static void Main(){

    bool a = true;

    goto mylabel;

    //Este comando não será executado
    if(a==true){
      Console.Write("Verdadeiro");
      goto end;
    }

     mylabel:
     Console.Write("Falso");

     //o label é sempre seguido por um comando, neste caso um comando vazio
     end:;
  }

}
```

Desvio (Jump)

Os comandos de desvio em C# são: *break*, *continue*, *goto*, *return* e *throw*. Basicamente, sua funcionalidade é desviar a execução do código para uma outra localização.

Adorado por alguns e odiado pela maioria dos programadores e especialistas, a instrução *goto* não foi abolidada da linguagem C#, tendo papel fundamental quando aplicada com o comando *switch*, que veremos mais adiante. O comando *goto* simplesmente executa um desvio de código através de um rótulo. Este desvio pode ser *top-down* (de cima para baixo) ou *bottom-up* (de baixo para cima).

```
//Loop infinito
endless:
goto endless;
```

O comando *return* é utilizado para devolver um valor e sair de uma função ou método chamado. Neste caso, o processamento é retornado para o chamador para a continuação do processamento. Em caso de funções que não retornam valor (*void*), o comando *return* poderá ser encontrado isolado, ou omitido se o mesmo deverá ser encontrado no fim da função. Os *snippets* abaixo exibem os casos mais comuns:

```
long z = Sum(10,20);
//continuação do programa...

private static long Sum(int x, int y){

//Soma os valores e retorna um long
return x+y;

}
```

```
private static bool boolFromint(int a){

//Verifica se o valor do inteiro é 0 e retorna false, senão retorna true
if(a==0)
  return false;
else
  return true;

}
```

```
private static void printf(string s){

//Imprime a string e retorna logo em seguida, cláusula return omitida
System.Console.Write(s);

}
```

O comando *throw* é utilizado para produzir uma exceção, e pode ser interceptado em tempo de execução pelo bloco *try/catch*.

```
private static double Division(int x, int y){

//Se o divisor for zero disparar a excessão da BCL DivideByZeroException
if(y==0) throw new DivideByZeroException();
return x/y;

}
```

Os comandos *break* e *continue* são utilizados com os comandos de iteração *switch*, *while*, *do*, *for* ou *foreach*. O comando *break* interrompe a execução do bloco destes comandos passando para próxima instrução ou bloco de execução. O comando *continue*, ao contrário do *break*, passa para a próxima iteração e verificação destes comandos.

```
int b=0;
for(int a = 0; a < 100; ++a){

//Loop infinito
while(true){
  if (++b==100) break; //Se b igual a 100, o break força a saída do loop while
}

b=0;
continue; //Passa para próxima iteração do comando for
System.Console.Write("a={0} b={1}",a,b); //Esta linha não é executada

}
//Continuação após o bloco for...
```

Seleção

Os comandos de seleção são utilizados na escolha de uma possibilidade entre uma ou mais possíveis. Os comandos *if* e *switch* fazem parte deste grupo.

Comando *if*

O comando *if* utiliza uma expressão, ou expressões, booleana para executar um comando ou um bloco de comandos. A cláusula *else* é opcional na utilização do *if*, no entanto, seu uso é comum em decisões com duas ou mais opções.

```
//if com uma única possibilidade. Exibe a string "Verdadeiro" no Console caso a
//expressão (a==true) seja verdadeira
if(a==true) System.Console.Write("Verdadeiro");
```

```
//if com uma única possibilidade. Exibe a string "Verdadeiro" no Console caso a
//expressão (a==true) seja verdadeira, senão exibe a string "Falso"
if(a==true)
  System.Console.Write("Verdadeiro");
else
  System.Console.Write("Falso");
```

Toda expressão do comando *if* deve ser embutida em parênteses (()) e possui o conceito de *curto-circuito* (*short-circuit*). Isto quer dizer que se uma expressão composta por *And* (*&&*), fornecer na sua primeira análise um valor booleano *false* (*falso*), as restantes não serão analisadas. Este conceito é válido para todas expressões booleanas. Por exemplo:

```
//&& (And). Somente a primeira função é executada
if(MyFunc() && MyFunc2());

//|| (Or). Ambas funções são executadas
if(MyFunc() || MyFunc2());

public static bool MyFunc(){ return false; }

public static bool MyFunc2(){ return true; }
```

Assim como outros comandos. O *if* também pode ser encontrado na forma aninhada.

```
if(x==1)
  if(y==100)
    if(z==1000)
      System.Console.Write("OK");
```

Porém, devido a característica de curto-circuito nas expressões, as linhas de cima podem e devem ser reescritas para:

```
if(x==1 && y==100 && z==1000) System.Console.Write("OK");
```

O comando *if* também pode ser encontrado num formato escada *if-else-if*, quando existem mais do que duas possibilidades. Porém, na maioria destes casos, se as expressões não forem compostas ou utilizarem de funções, a cláusula *switch* substitui este tipo de construção.

```
using System;

class Ifs{

  public static void Main(){

    char chOpt;

    Console.WriteLine("1-Inserir");
    Console.WriteLine("2-Atualizar");
    Console.WriteLine("3-Apagar");
    Console.WriteLine("4-Procurar");
    Console.Write("Escolha entre [1] a [4]:");

    //Verifica se os valores entrados esta entre 1 e 4
    //caso contrário pede reentrada
    do{

      chOpt = (char)Console.Read();

    }while(chOpt<'1' || chOpt>'4');

    if(chOpt=='1'){
      Console.WriteLine("Inserir...");
      //InsertFunction();
    }
    else if(chOpt=='2'){
      Console.WriteLine("Atualizar...");
      //UpdateFunction();
    }
    else if(chOpt=='3'){
      Console.WriteLine("Apagar...");
      //DeleteFunction();
    }
    else{
      Console.WriteLine("Procurar...");
      //FindFunction();
    }

}
```

O comando *if* com a cláusula *else* única pode ser encontrado em sua forma reduzida com operador ternário representado por interrogação (?:). É chamado de operador ternário por possuir 3 expressões: a primeira refere-se a condição boolena, a segunda se a condição é verdadeira e a terceira se a condição é falsa.

```
int x;

if(f==true)
  x = 100;
else
  x = 1000;
```

As linhas acima podem ser substituídas por:

```
int x = f==true?100:1000;
```

Comando *switch*

O comando *switch* utiliza o valor de uma determina expressão contra uma lista de valores constantes para execução de um ou mais comandos. Os valor constante é tratado através da cláusula *case* e este pode ser númerico, caracter ou *string*. A cláusula *default* é utilizada para qualquer caso não interceptado pelo *case*. O exemplo abaixo implementa a versão com o comando *switch* do exemplo, previamente mostrado com o comando *if*:

```
using System;

class Switchs{

  public static void Main(){

    char chOpt;

    Console.WriteLine("1-Inserir");
    Console.WriteLine("2-Atualizar");
    Console.WriteLine("3-Apagar");
    Console.WriteLine("4-Procurar");
    Console.Write("Escolha entre [1] a [4]:");

    //Verifica se os valores entrados esta entre 1 e 4
    //caso contrário pede reentrada
    do{

      chOpt = (char)Console.Read();

    }while(chOpt<'1' || chOpt>'4');

    switch(chOpt){
    case '1':
      Console.WriteLine("Inserir...");
      //InsertFunction();
      break;
    case '2':
      Console.WriteLine("Atualizar...");
      //UpdateFunction();
      break;
    case '3':
      Console.WriteLine("Apagar...");
      //DeleteFunction();
      break;
    default:
      Console.WriteLine("Procurar...");
      //FindFunction();
    }

}
```

Para compilar o exemplo acima, no prompt, digite *csc Switchs.cs*. Execute o programa digitando *Switchs*. A Figura 2, mostra compilação e execução da aplicação em C#.

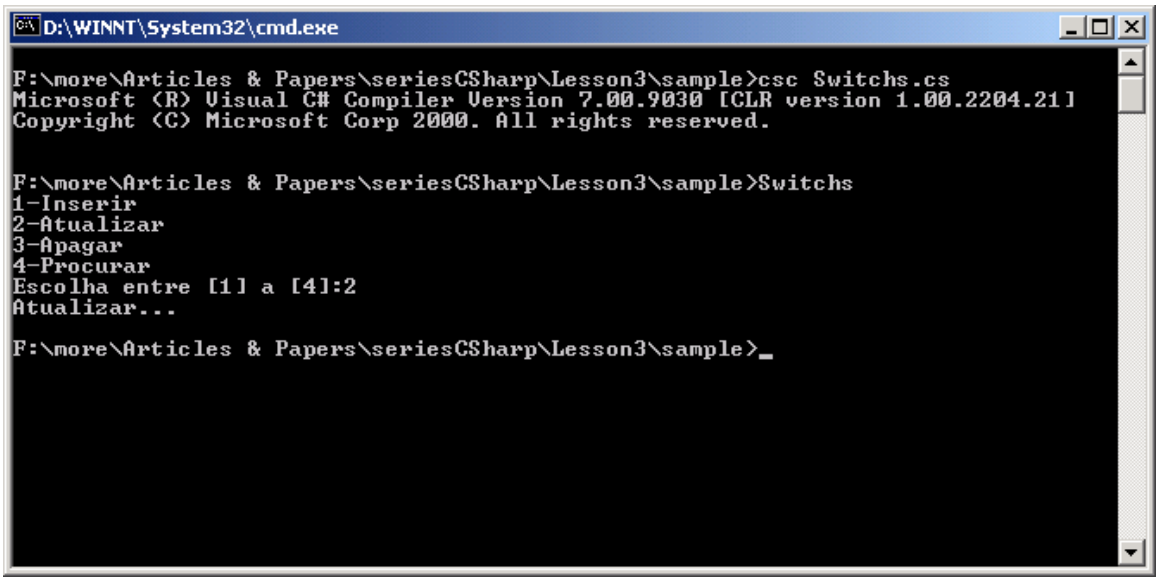

Figura 8: Compilação e Execução do exemplo Switchs

No entanto, o comando *switch* não herda as características do mesmo em C ou C++, uma cláusula *case* força um comando de desvio como *break, goto* ou *return* assim as outras cláusulas *case* não serão processadas, o *break* em C# não causa o efeito *fall through.* Se o programa precisar tratar mais do que uma cláusula *case* com códigos distintos no mesmo *switch,* o comando *goto* deverá ser utilizado. Uma ou mais cláusulas *case* podem ser encontradas seguidamente quando mais do que uma opção é permitida para um comando ou bloco de comandos. O exemplo abaixo apresenta essa condição:

```
switch(sProduct){
  case "Windows 2000":
  case "Windows NT":
    System.Console.Write("Sistema Operacional");
    break;
  case "MSDE":
    System.Console.Write("Mecanismo Simplificado");
    goto case "SQL Server";
  case "SQL Server":
    System.Console.Write("Banco de Dados");
}
```

Assim como o comando *if* é possível encontrar o comando *switch* em sua forma aninhada.

```
switch(x){
  case 10:
    switch(y){
      case 100:
      case 1000:
    }
    break;
  case 100:
    break;
}
```

Iteração ou Loop

Conhecidos como laço ou loop, os comandos de iteração executam repetidamente um comando ou bloco de comandos, a partir de uma determinada condição. Esta condição pode ser pré-definida ou com final em aberto. Em C#, fazem parte dos comandos de iteração: *while*, *do*, *for* e *foreach*.

Comando *for*

O comando *for* possui 3 declarações opcionais, separadas por ponto e vírgula (;), dentro dos parênteses: inicialização, condição e a iteração. Em cada parâmetro, mais de uma expressão pode ser encontrada separada por vírgula.

```
for(int x=0; x < 100; ++x) System.Console.WriteLine(x);
```

```
for(;;) System.Console.WriteLine("Hello, World!");
```

```
for(int y=100, int x = 0; x < y; ++x, --y) System.Console.WriteLine(y);
```

Quando a cláusula *for* é processada pela primeira vez, se presente, a expressão ou expressões da declaração inicializadora são executadas na ordem que elas estão escritas, este passo ocorre apenas uma vez. Se a declaração condicional estiver presente, será avaliada, caso contrário o *for* assume o valor verdadeiro (*true*). Na avaliação, se o valor obtido for verdadeiro (*true*) o comando ou bloco de comandos associados serão executados, ao seu final a terceira declaração ou declaração de iteração é processada e, então, novamente a declaração condicional é processada. Este fluxo ocorre continuamente até que a declaração condicional seja avaliada como falsa (*false*) ou o comando *break* seja encontrado, como visto anteriormente. O comando *continue* força uma nova iteração.

```
using System;

class Fibonacci{

  public static void Main(){

    int iVezes;

    Console.Write("Entre de 1 a 100 para o n° de elementos a exibir na sequência
de Fibonacci:");

    //Verifica se os valores entrados esta entre 1 e 100
    //caso contrário pede reentrada
    do{

      iVezes = Console.ReadLine().ToInt32();

    }while(iVezes<1 || iVezes>100);

    //Cria o vetor dinâmicamente
    int[] iSeq = new int[iVezes];

    iSeq[0] = 1;

    //Preenche o vetor
    if(iVezes > 1){

      iSeq[1] = 1;
```

```
      for(int a=2; a < iVezes; ++a)
      iSeq[a] = iSeq[a-1] + iSeq[a-2];

    }

    //Exibe o vetor
    for(int a=0; a < iVezes; ++a){

      Console.Write(iSeq[a]);
      Console.Write(" ");

    }

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc Fibonacci.cs*. Execute o programa digitando *Fibonacci*. A Figura 3, mostra compilação e execução da aplicação em C#.

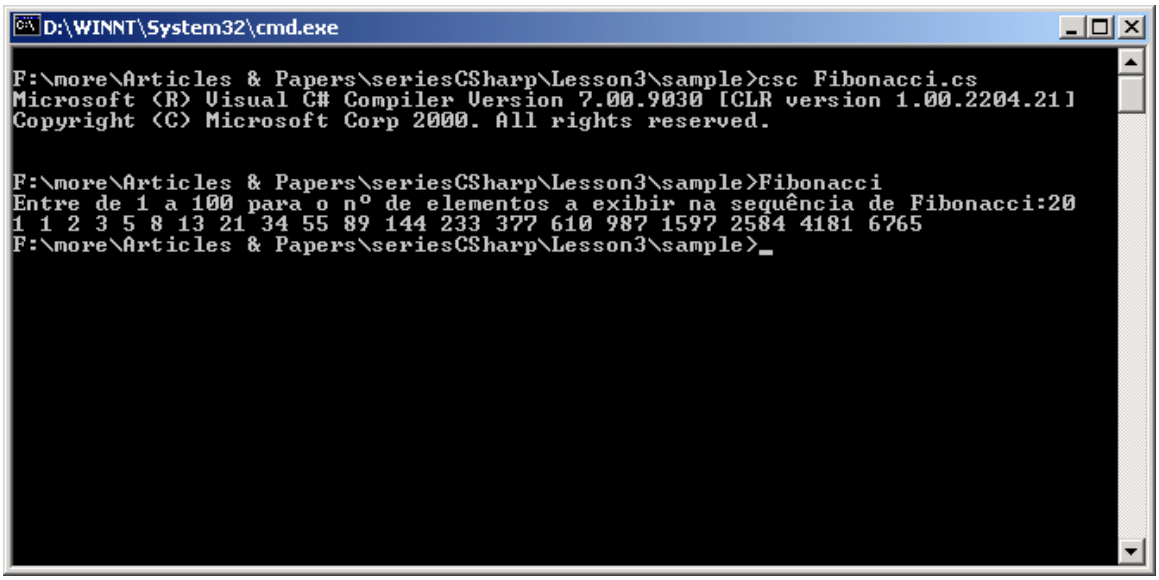

Figura 9: Compilação e Execução do exemplo Fibonacci

Comando *foreach*

O comando *foreach* enumera os elementos de uma coleção. O código abaixo implementa a funcionalidade do exemplo anterior:

```
using System;

class Fibonacci{

  public static void Main(){

    int iVezes;

    Console.Write("Entre de 1 a 100 para o nº de elementos a exibir na sequência
de Fibonacci:");
```

```
    //Verifica se os valores entrados esta entre 1 e 100
    //caso contrário pede reentrada
    do{

      iVezes = Console.ReadLine().ToInt32();

    }while(iVezes<1 || iVezes>100);

    //Cria o vetor dinâmicamente
    int[] iSeq = new int[iVezes];

    iSeq[0] = 1;

    //Preenche o vetor
    if(iVezes > 1){

      iSeq[1] = 1;
      for(int a=2; a < iVezes; ++a)
      iSeq[a] = iSeq[a-1] + iSeq[a-2];

    }

    //Exibe o vetor
    foreach(int a in iSeq){

      Console.Write(a);
      Console.Write(" ");

    }

  }

}
```

Os vetores em C# herdam da classe *System.Array* do .NET Framework e implementam a interface *IEnumerable* que possui o método *GetEnumerator* que retorna a interface *IEnumerator* que possui 3 membros: a propriedade *Current* que retorna o objeto atual, o método *MoveNext* que pula para o próximo elemento e o método *Reset* que reinicializa o posicionamento do elemento atual. Qualquer interface ou classe que implemente *IEnumerable* e *IEnumerator* pode utilizar o comando *for each*.

O comando *foreach* compacta a sequência abaixo:

```
System.Collections.IEnumerator ienumSeq = iSeq.GetEnumerator();

while(ienumSeq.MoveNext()){

  System.Console.WriteLine(ienumSeq.Current);

}
```

```
foreach(int a in iSeq){

  System.Console.WriteLine(a);

}
```

Comandos *do* e *while*

Os comandos *do* e *while* têm características semelhantes. Ambos executam condicionalmente um comando ou bloco de comandos. No entanto, o comando *do* pode ser executado uma ou mais vezes e o comando *while* pode ser executado nenhuma ou mais vezes, isto ocorre porque a expressão condicional do comando *do* é encontrada no final do bloco.

```
int a = 0;
bool f = true;

while(f){

  if(++a==100) f = true;
  System.Console.WriteLine(a);

}
```

```
int a = 0;
bool f = true;

do{

  if(++a==100) f = true;
  System.Console.WriteLine(a);

} while(f);
```

Assim como para os comandos *for* e *foreach*, as cláusulas *break* e *continue* podem ser utilizadas para interferir no fluxo de execução.

Outros comandos

Outros comandos, com finalidades distintas e não agrupados nos itens citados anteriormente, são: *try*, *catch*, *finally*, *checked*, *unchecked*, *unsafe*, *fixed* e *lock*.

Os comandos *try*, *catch* e *finally* são utilizados na intercepção e tratamento de exceção em tempo de execução.

```
using System;

class try_catch_finally{

  public static void Main(){

    try{

      Console.WriteLine("Bloco try");
      throw new NullReferenceException();

    }

    catch(DivideByZeroException e){

       Console.WriteLine("Bloco catch #1. Mensagem: {0}",e.Message);

     }

    catch(NullReferenceException e){
```

```
       Console.WriteLine("Bloco catch #2. Mensagem: {0}",e.Message);
     }
    catch(Exception e){
       Console.WriteLine("Bloco catch #3. Mensagem: {0}",e.Message);
    }
    finally{
      Console.WriteLine("Bloco finally");
    }
  }
}
```

Os comandos *checked* e *unchecked*, tratam de *overflow* aritmético. O comando *checked* dispara a exceção *OverflowException* e o comando *unchecked* trunca o valor. O parâmetro /*checked*+ do compilador trata o *overflow* como *checked* e o parâmetro /*checked*- do compilador trata o *overflow* como *unchecked*, este é o padrão se não especificado.

```
using System;

class Overflows{

  public static void Main(){

    try{

      short a = 32767;
      short b = (short)(a + 1);

      Console.Write("{1} + 1 = {0}",b,a);

    }

    catch(OverflowException e){

       Console.WriteLine("Mensagem: {0}",e.Message);

     }

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc /checked+ Overflows.cs* para a condição *checked*. Execute o programa digitando *Overflows*. Depois no prompt, digite *csc /checked- Overflows.cs* para a condição *unchecked*. Execute o programa digitando *Overflows*. A Figura 4, mostra as compilações e execuções da aplicação em C#.

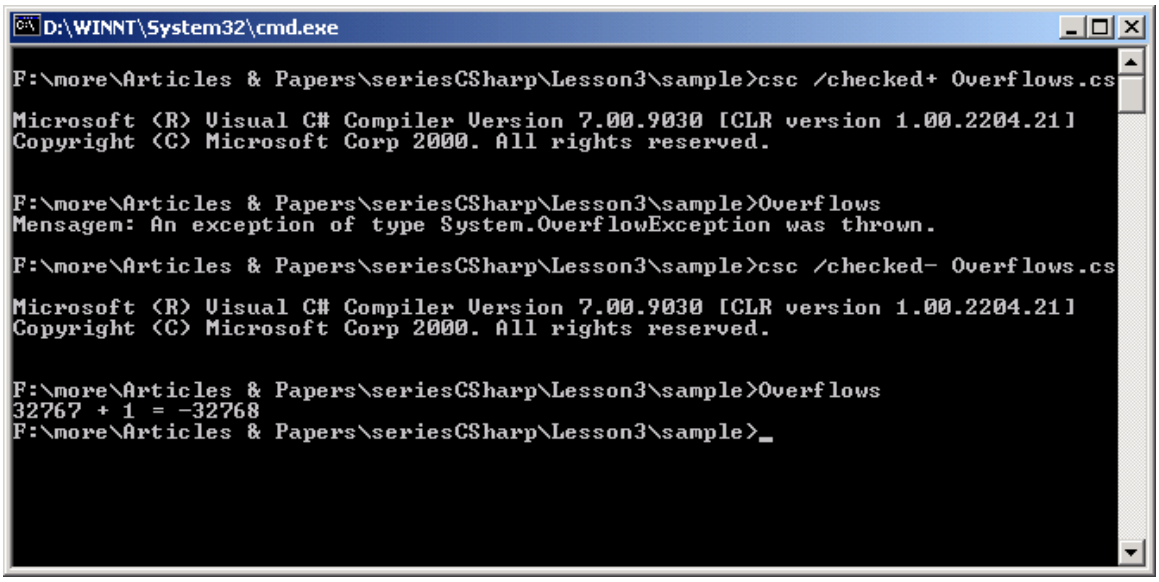

Figura 10: Compilações e Execuções do exemplo Overflows

Os comandos *checked* e *unchecked* podem ser utilizados dentro do programa para alterar a condição especificada na compilação.

```
using System;

class Overflows2{

  public static void Main(){

    try{

      short a = 32767;
      short b = unchecked((short)(a + 1));

      Console.WriteLine("unchecked: {1} + 1 = {0}",b,a);

      short c = 32767;
      short d = checked((short)(c + 1));

      Console.WriteLine("checked: {1} + 1 = {0}",d,c);

    }

    catch(OverflowException e){

       Console.WriteLine("checked: Mensagem - {0}",e.Message);

     }

  }

}
```

Os comandos *unsafe* e *fixed* é utilizado na operação com ponteiros. O parâmetro */unsafe+* do compilador torna todo o código apto ao tratamento de ponteiros. Por exemplo:

```
using System;

class Pointers{

  unsafe public static void Process(int[] a){

    fixed(int* pa = a){

      for(int i=0;i<a.Length;++i)
        Console.Write("{0} ",*(pa+i));

    }

  }

  public static void Main(){

    int[] arr = {1,2,3,4,5,6,7,8,9,0};

    unsafe Process(arr);

  }

}
```

O comando *lock* utiliza *critical section* para bloqueio de acesso consecutivo dentro de uma thread.

```
using System;
using System.Threading;

class Locks{

  static int x=0;

  public static void ThreadProc(){

    lock(typeof(Locks)){
      x++;
    }

    Console.WriteLine("x = {0}",x);

  }

  public static void Main(){

    for(int a=0; a<10; ++a){
      Thread t = new Thread(new ThreadStart(ThreadProc));
      t.Start();
    }

  }

}
```

Conclusão

Neste artigo, são apresentados os comandos do fluxo de programa da linguagem C#. São apresentadas a declaração e a utilização deles. Os comandos de declaração, rótulo, desvio, seleção, iteração e outros são aplicados nos exemplos. Outras opções do compilador, bem como novas classes do .NET Framework são usadas.

Links

http://msdn.microsoft.com/vstudio/nextgen/default.asp
http://www.microsoft.com/net/default.asp
http://msdn.microsoft.com/net/
http://gotdotnet.com/

Conhecendo C# - Lição 4 : Operadores

por Fabio R. Galuppo

C# é uma linguagem muito rica em operadores. Estes representados por símbolos são utilizados na construção de expressões. A sintaxe de expressão do C# é baseada na sintaxe do C++. Os operadores são categorizados em diversas funcionalidades. A tabela 1 apresenta essas divisões.

Tabela 4: Operadores do C#

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

Quando uma expressão possui múltiplas operações, a precedência dos operadores é levada em consideração na avaliação da mesma. Normalmente, as expressões são avaliadas da esquerda para direita, exceto para operações de atribuição e condicional, porém a precedência pode ser alterada através do uso do parenteses.

```
x = 10 + 100 * 1000   // x = 100010
x = (10 + 100) * 1000 // x = 110000
```

A tabela 2 abaixo relaciona, na ordem do maior para o menor, os operadores em relação a precedência.

Tabela 5: Precedência dos operadores

Categoria	Operadores
Primário	(x) x.y f(x) a[x] x++ x—new typeof sizeof checked uncheked
Unário	+ - ! ~ ++x –x (T)x
Multiplicativo	* / %
Aditivo	+ -
Shift	<< >>
Relacional	< > <= >=< > <= >= is
Comparação	== !=
Lógico AND	&
Lógico XOR	^
Lógico OR	|
Condicional AND	&&
Condicional OR	||
Condicional	?:
Atribuição	= *= /= %= += -= <<= >>= &= ^= |=

Operadores Aritméticos

Os operadores aritméticos são utilizados na maioria das expressões para execução de cálculos. Numa expressão, eles podem produzir resultados fora da faixa de valores. Neste caso, uma excessão como *OverflowException* é gerada, no entanto, este comportamento pode ser alterado pelo bloco *unchecked* ou através da compilação. Os tópicos abaixo devem ser levados em consideração quando esses operadores são utilizados:

- Overflow em operações aritméticas com inteiros geram a exceção *OverflowException* ou discartam o bit mais significativo do resultado;
- Divisão por zero em operações aritméticas com inteiros sempre geram a exceção *DivideByZeroException*;
- Operações aritméticas com valores de ponto flutuante são baseadas em ponto flutuante IEEE 754, portanto, overflow e divisão por zero não geram exceção. Elas são representadas por infinito ou *NaN (Not a Number)*;
- Overflow em operações aritméticas com decimais sempre geram a excessão *OverflowException*;
- Divisão por zero em operações aritméticas com decimais sempre geram a excessão *DivideByZeroException*;

Os operadores unários + e – são utilizados para representar se o número é positivo ou negativo, respectivamente.

```
x = +1000 // x = 1000
x = -1000 // x = -1000
```

Os operadores binários +, -, *, / e % são utilizados nas expressões para execução de cálculos tais como soma, subtração, multiplicação, divisão e sobra. O operador binário + quando utilizado entre strings representam concatenação. No entanto, quando existem strings e números na expressão e nenhuma operação de cast for executada a operação é tratado como concatenação. O operador binário % é computado através da fórmula *dividendo – (dividendo / divisor) * divisor*. Os exemplos abaixo ilustram essas condições

```
string x = "Hello" + "World"      // x = "HelloWorld"

string x = "Valor = " + 100       // x = "Valor = 100"

int x = 1000 % 11                 // x = 10

int x = 1000 - ( 1000 / 11 ) * 11 // x = 10
```

O código abaixo utiliza os operadores aritméticos. Note a utilização e recebimento de argumentos através da linha de comando. O entry-point Main permite ser passado um vetor de strings como parâmetro. O método *Length* conta o número de parâmetros passado, ignorando o executável. Como todo vetor o primeiro parâmetro é representado pelo índice

zero, por exemplo *args[0]*. A váriavel *args* não é uma palavra-chave, portanto, esta pode ser alterada:

```
using System;

class Arithmetics{

  public static void Main(string[] args){

    //Verifica o número de argumentos entrados
    if(args.Length == 3){
      int x=0,y=0;

      //Convertem os valores dos argumentos 2 e 3 para inteiro 32-bit
      //Se ocorrer algum erro o modo de utilização
      try{
        x = args[1].ToInt32();
        y = args[2].ToInt32();
      }
      catch{
        usage();
        return;
      }

      //Efetua a operação selecionada no primeiro argumento
      switch(args[0]){
      case "+":
        Console.Write("Valor da soma = {0}", x+y);
        break;
      case "-":
        Console.Write("Valor da subtração = {0}", x-y);
        break;
      case "/":
        Console.Write("Valor da divisão = {0}", x/y);
        break;
      case "*":
        Console.Write("Valor da multiplicação = {0}", x*y);
        break;
      case "%":
        Console.Write("Valor da sobra = {0}", x%y);
        break;
      default:
        usage();
  }
  else{
    usage();
  }

}

public static void usage(){

  //Modo de utilização
  Console.WriteLine("Modo de usar: Arithmetics operador valor1 valor2");
  Console.WriteLine("Ex.: Arithmetics + 100 200");

}

}
```

Para compilar o exemplo acima, no prompt, digite *csc Arithmetics.cs*. Execute o programa digitando *Arithmetics operador valor1 valor2(ex.: Arithmetics + 100 200)*. A Figura 1, mostra compilação e execução da aplicação em C#.

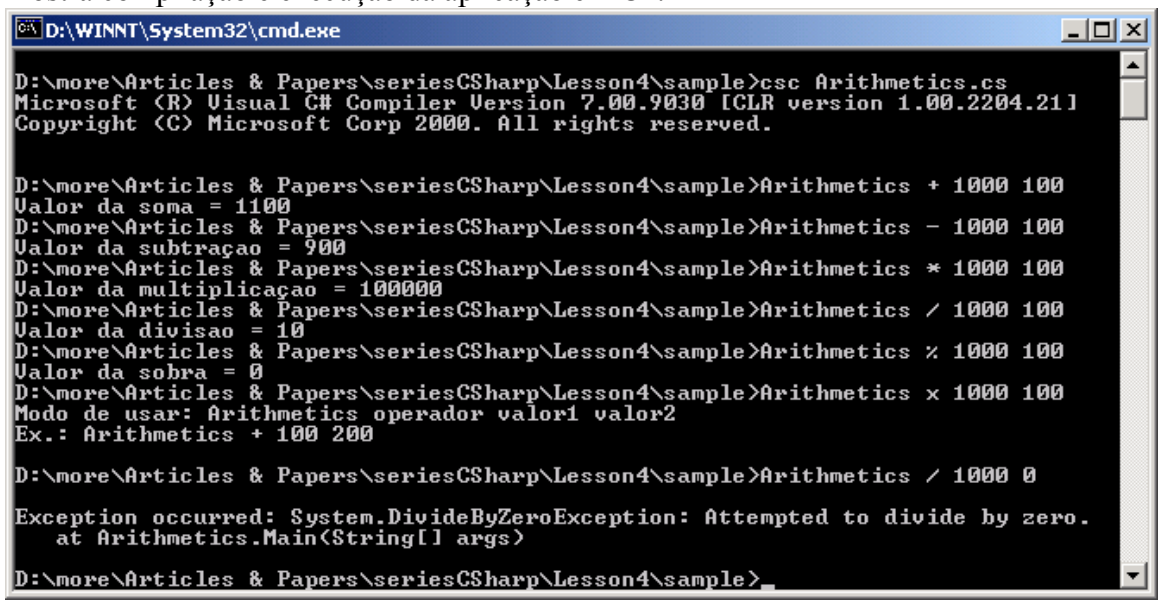

Figura 11: Compilação e Execução do exemplo Arithmetics

Operadores Shift

Os operadores << e >>, executam rolagem à esquerda ou à direita, respectivamente. Quando o operador << é utilizado, o bit mais significativo é descartado e o bit menos significativo recebe um zero. Quando o operador >> é utilizado em operações com *uint* ou *ulong* o bit menos significativo é descartado e o bit mais significativo recebe um zero, no caso de *int* e *long* o bit menos significativo é descartado e o bit mais significativo recebe um zero se não for positivo e um se for negativo. As expressões que utilizam esses operadores utilizam um valor inteiro a direita para indicar o número de rolagem.

```
Console.WriteLine(1 << 2);   // 4 -> 1 representa 001 4 representa 100
Console.WriteLine(4 >> 2);   // 1
Console.WriteLine(-1 << 2);  // -4
Console.WriteLine(-4 >> 2);  // -1
Console.WriteLine(1 << 32);  // 1 -> 1 (int) inteiro 32-bit
Console.WriteLine(1L << 32); // 4294967296 -> 1 (long) inteiro 64-bit
```

Operadores Incremento e Decremento

Os operadores ++ e -- aumentam ou diminuem por um o valor correspondente. O ponto chave é que se o operador for utilizado à esquerda da variável, ou seja prefixado, o valor é adicionado ou subtraído de um antes de sua utilização.

```
int x = 1000; // x = 1000
x++;          // x = 1001
int y = x++;  // x = 1002 , y = 1001
x--;          // x = 1001
y = --x;      // x = 1000 , y = 1000
```

```
++x;          // x = 1001
--x;          // x = 1000
y = ++x;      // x = 1001 , y = 1001
```

Operadores Lógico, Relacional e Condicional

Esses operadores são utilizados em expressões onde o resultado retornado ou a característica é booleana.

O operador de negação ! retorna o complemento de um valor booleano.

```
bool x = true
bool y = !x // y = false

if(!y) System.Console.Write("y é verdadeiro")
```

Os operadores relacionais ==, !=, <, >, <=, >=, resultam em um valor booleano e representam *igual*, *não igual* ou *diferente*, *menor*, *maior*, *menor ou igual* e *maior ou igual*, respectivamente. Por exemplo, *a == b* quer dizer se *a* for igual a *b*, isto é totalmente válido na expressão com tipos primitivos(*value types*), tais como *int*, *long*, *char*, entre outros. Porém o comportamento dos operadores == e != são diferenciados quando utilizado entre *structs (value types)* e *classes* (*reference types)*. Para structs a comparação deve levar em consideração todos os campos da estrutura. Para classes a comparação é efetuada pelo endereço, ou referência, da classe. O único *reference type* que compara o valor e não a referência é a string ou a classe System.String, pois os operadores == e != são sobrecarregados. A sobrecarga pode alterar o comportamento padrão dos operadores.

```
using System;

class CX{

  public int x;
  public int y;

  int m_z;

  public int z{

    set{ m_z = value; }
    get{ return m_z;  }

  }

}

struct SX{

  public int x;
  public int y;

}

class CompareTypes{

  public static void Main(){

    CX cx1 = new CX();
    CX cx2 = new CX();
```

```
SX sx1, sx2;

sx2.x = sx1.x = cx2.x = cx1.x = 100;
sx2.y = sx1.y = cx2.y = cx1.y = 200;
cx2.z = cx1.z = 300;

//Comparando tipos primitivos
Console.WriteLine("\nTipos Primitivos");

if(sx1.x==sx2.x)
  Console.WriteLine("sx1.x é igual a sx2.x");
else
  Console.WriteLine("sx1.x é diferente de sx2.x");

if(cx1.y==cx2.y)
  Console.WriteLine("cx1.y é igual a cx2.y");
else
  Console.WriteLine("cx1.y é diferente de cx2.y");

if(sx1.x>=sx1.y)
  Console.WriteLine("sx1.x é maior ou igual a sx1.y");
else
  Console.WriteLine("sx1.x é menor que sx1.y");

if(cx1.x<=cx1.y)
  Console.WriteLine("cx1.x é menor ou igual a cx1.y");
else
  Console.WriteLine("cx1.x é maior que cx1.y");

if(sx1.x>sx1.y)
  Console.WriteLine("sx1.x é maior que sx1.y");
else
  Console.WriteLine("sx1.x é menor ou igual a sx1.y");

if(cx1.x<cx1.y)
  Console.WriteLine("cx1.x é menor que cx1.y");
else
  Console.WriteLine("cx1.x é maior ou igual a cx1.y");

if(sx1.x!=sx1.y)
  Console.WriteLine("sx1.x é diferente a sx1.y");
else
  Console.WriteLine("sx1.x é igual a sx1.y");

//Comparando classes
Console.WriteLine("\nClasses");

if(cx1==cx2)
  Console.WriteLine("cx1 é igual a cx2");
else
  Console.WriteLine("cx1 é diferente de cx2");

cx2 = cx1;

if(cx1==cx2)
  Console.WriteLine("cx1 é igual a cx2");
else
  Console.WriteLine("cx1 é diferente de cx2");

//Comparando structs
Console.WriteLine("\nStructs");
```

```
    if((sx1.x==sx2.x) && (sx1.y==sx2.y))
      Console.WriteLine("sx1 é igual a sx2");
    else
      Console.WriteLine("sx1 é diferente de sx2");
  }

}
```

Os operadores lógicos &, |, ^, ~, &&, ||, *true* e *false* representam as operações bit-a-bit AND, bit-a-bit OR, bit-a-bit XOR, bit-a-bit NOT, AND, OR, verdadeiro e falso. Os operadores && e || possuem o efeito de curto-circuito na avaliação da expressão. O efeito curto-circuito despreza o restante da avaliação da expressão quando numa operação AND o item avaliado é falso e numa operação OR o item avaliado é verdadeiro.

```
true & false  // false
0x1 & 0x10    // 0x0
```

```
true | false // true
0x1 & 0x10   // 0x11
```

```
false ^ true // false
0x1 ^ 0x10   // 0x11
```

```
~0x000000ff  // 0xffffff00
```

```
int x = 10, y = 100;

//A primeira avaliação (x==100) retorna false portanto o restante é desprezado
if(x==100 && y==10) System.Console.Write("Verdadeiro");

//A primeira avaliação (x==10) retorna true portanto o restante é desprezado
if(x==10 || y==100) System.Console.Write("Verdadeiro");
```

Operadores de Atribuição

Estes operadores, divididos entre simples e compostos, são utilizados na designação de um valor para uma variável. O operador = representa a atribuição simples, ou seja uma variável do lado esquerdo recebe o contéudo de um valor, de uma variável ou do resultado de uma expressão do lado direito.

```
int x = 10;
int y = x;
int z = x + y;
```

Os operadores +=, -=, *=, /=, %=, &=, |=, ^=, <<= e >>= representam a atribuição composta, que normalmente atuam como um atalho na construção de uma expressão.

```
x = x + 10; //Pode ser escrito como:
x+= 10;     // x <op>= <valor>
```

Em alguns casos são utilizadas as operações de *cast* em conjunto a essas operações.

```
double x = 10.111;
float z = (float)(x * 10);
float k+= (float)x;
```

Operadores de Tipo

Os operadores *typeof* e *is* são utilizados para obter o tipo do objeto em tempo de execução. O operador *typeof* retorna o tipo de um objeto que é uma instância de System.Type. O operador is verifica se um objeto é compatível com um determinado tipo, isto pode ser uma classe, uma interface, uma estrutura ou um tipo primitivo.

```
using System.WinForms;
using System.Reflection;

class SimpleReflections{

  public static void Main(){

    System.Type t = typeof(MessageBox);

    object   o  = System.Activator.CreateInstance(t);
    object[] op = {"Hello, World!"};
    t.InvokeMember("Show",BindingFlags.InvokeMethod,null,o,op);

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc /r:System.WinForms.dll SimpleReflections.cs*. Execute o programa digitando *SimpleReflections*. A Figura 2, mostra compilação e execução da aplicação em C#. Algumas novidades são mostradas neste exemplo, uma delas chama-se *Reflection* que tem suas classes associadas ao namespace *System.Reflection*. Outra seria o uso dos formulários Windows, que estão associados ao namespace *System.WinForms* e encontram-se no assembly *System.WinForms.dll*. Quando um recurso não é encontrado na *mscorlib.dll*, como é o caso da *MessageBox*, a mesma deve ser indicada como argumento */r:<assembly>* do compilador , por exemplo */r:System.WinForms.dll*.

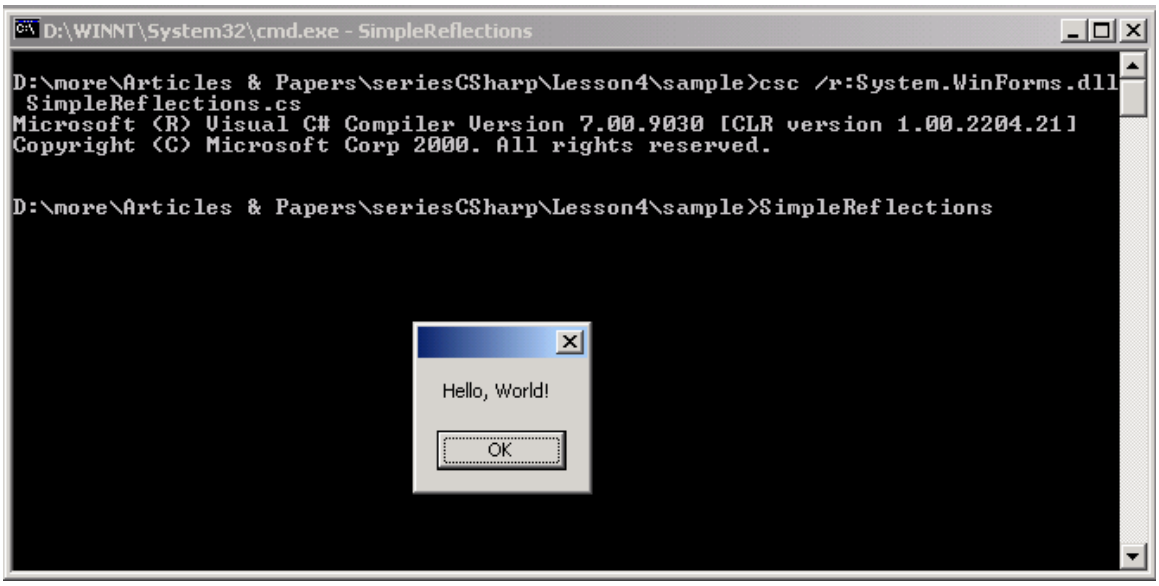

Figura 12: Compilação e Execução do exemplo SimpleReflections

O operador *sizeof* é o velho conhecido dos desenvolvedores C. Ele retorna o tamanho em bytes de um *value type*. O *sizeof* não retorna valor para *reference types*, só pode ser utilizado dentro de um bloco *unsafe* e não pode ser sobrecarregado.

```
using System;
struct SX{ public int x, y, z; }

class Sizeofs{

  public static void Main(){

    unsafe{

      Console.WriteLine("O tamanho do byte  em bytes é {0}.", sizeof(byte));
      Console.WriteLine("O tamanho do short em bytes é {0}.", sizeof(short));
      Console.WriteLine("O tamanho do int   em bytes é {0}.", sizeof(int));
      Console.WriteLine("O tamanho do long  em bytes é {0}.", sizeof(long));
      Console.WriteLine("O tamanho do SX    em bytes é {0}.", sizeof(SX));

    }

  }

}
```

Outras palavras-chave que atuam com operações com tipos são *as*, *new* e *stackalloc*. O operador *as* é utilizado na execução de conversão entre *reference types* compatíveis, senão o valor *null* é retornado, isto é o mesmo que: *expressão is tipo ? (tipo)expressão : (tipo) null*. O new, em sua forma de operador, a partir de uma classe cria o objeto na *heap (Managed Heap)* e invoca seu construtor. O stackalloc aloca um bloco de memória na stack, o endereço desse bloco é retornado em um ponteiro, em consequência disso deve ser encontrado dentro de um bloco *unsafe*.

```
string x = "Isto é uma string";
object y = x as System.Collections.Stack; // y = null
```

```
object z = x as string;                    // z = "Isto é uma string"
```

```
System.Collections.Stack stack = new System.Collections.Stack()
```

```
using System;

class StackAllocs{

  public static void Main(string[] args){

   int count = args[0].ToInt32();

   unsafe{

      int* pi = stackalloc int[count];
      for(int a = 0; a < count; ++a) pi[a] = a+1;
      for(int a = 0; a < count; ++a,++pi) Console.WriteLine(*pi);

    }

  }

}
```

Operadores de Controle de Excessão e Overflow

As palavras-chave checked e unchecked podem ser utilizadas para disparar uma excessão ou truncar o valor quando um overflow ocorrem num cálculo aritmético. Estes são usados na forma de bloco ou de expressão. O comportamento padrão é equivalente a unchecked, no entanto é possível alterá-lo com o parâmetro */checked+* para *checked* ou */checked-* para *unchecked*, mas se encontrado *checked* ou *unchecked* no código-fonte o comportamento padrão é ignorado para do trecho onde o bloco ou a expressão é presente.

É importante saber que as operações que fazem verificação de overflow:

- ++ -- -(unário) + -* /
- Conversões númericas explícitas entre tipos integrais

```
checked{
  byte x = 255;
  System.Console.Write(++x);
}
```

```
unchecked{
  byte x = 255;
  System.Console.Write(++x);
}
```

```
byte x = 255;
Console.Write(checked(++x));
```

```
byte x = 255;
Console.Write(unchecked(++x));
```

```
class Overflows{

  public static void Main(){
```

```
    byte x = 255;
    System.Console.Write(++x);

  }

}
```

Se o código acima for compilado com a linha de instrução *csc Overflows.cs /checked+*, sua execução disparará a excessão *System.OverflowException*. Se o código acima for compilado com a linha de instrução *csc Overflows.cs /checked-*, sua execução mostrará o valor 0.

Operadores de Indireção e Ponteiros

Os operadores *, ->, [] e & são utilizados para manipulação de ponteiros e endereços, mas seu uso é somente permitido dentro do bloco *unsafe*. O exemplo abaixo mostra o uso destes operadores:

```
struct SX{ public int x; }

class Pointers{

  public static void Main(){

    SX sx;
    sx.x = 0;

    unsafe{

      SX* psx = &sx;
      psx->x = 10;

    }

    System.Console.Write(sx.x);

  }

}
```

Outros Operadores

O operador . representa acesso a membro de uma classe, interface, enumeração ou estrutura e também a acesso a qualificador (*qualified name*) de namespace ou interface.

```
System.Console.Write("Uso do . com namespace e class");
```

```
class CX{ public int x; }

CX cx;
cx.x;
```

```
struct SX{ public int x; }

SX sx;
```

```
sx.x;
```

O operador [] é utilizado juntamente com vetores, *indexers* e atributos.

```
x[1] = 1000;
```

O operador () é utilizado para determinar a ordem de processamento da expressão, bem como serve para indicação de conversão explícita (*cast*).

```
byte = (byte)100L;
int x = ( 1 + 2 ) * 3;
```

O operador ?: é usado em conjunto com uma expressão de condição para retornar uma de duas possibilidades. Normalmente é utilizada para simplicar a instrução *if-else* simples de atribuição a uma mesma variável. Porém esta não é uma prática muito utilizada.

```
if(a==1 && b ==2) x = 100; else x =1000;
x = (a==1 && b ==2) ? 100 : 1000;
```

O operador + e – é utilizado também juntamente com *delegates*.

```
class Delegates{

  delegate void Trace(string s);

  public static void Main(){

    Trace trace = new Trace(TraceConsole);
    Trace trace2 = new Trace(TraceMsgBox);
    trace += trace2;
    trace("Isto é um delegate com 2 notificações!");
    trace -= trace2;
    trace("Isto é um delegate com 1 notificação!");

  }

  static void TraceConsole(string st){

    System.Console.WriteLine(st);

  }

  static void TraceMsgBox(string st){

    System.WinForms.MessageBox.Show(st);

  }

}
```

Compile o código acima utilizando *csc Delegates.cs /r:System.WinForms.dll*.

Operadores Built-In

Para operações númericas o C# possui as conversões implícitas que podem ser consideradas como operadores built-in para o int, uint, long, ulong, float, double e decimal. Essas

conversões são feitas automaticamente do menor tipo para o maior tipo, caso contrário a conversão explícita deve ser aplicada.

```
short x = 100;
int   z = x;        // operador built-in int
short k = (short)z; // necessidade de cast
```

Conclusão

Neste artigo, são apresentados os operadores da linguagem C#. É mostrado a declaração e utilização deles. Vários exemplos são aplicados com o uso dos operadores, inclusive mais opções de compilação, da linguagem e bibliotecas também são empregados.

Links

http://www.microsoft.com/brasil/net/
http://msdn.microsoft.com/net/
http://communities.msn.com.br/CeSharp

Conhecendo C# - Lição 5 : Excessões

por Fabio R. Galuppo

No mundo dos *frameworks* e linguagens de programação, as excessões, ações que causam anômalias nas aplicações são tratadas de diversas formas. Algumas utilizam códigos de retorno, como por exemplo COM que utiliza o *HRESULT*, tratamento de excessões não estruturadas, como no caso do Visual Basic que utiliza *ON ERROR* e o objeto *ERR*, valor do erro retornado assim como no Win32, tratamento de excessões estruturadas assim como C++, *true* ou *false*, entre outros. Apesar dessas diversificações, o .NET Framework elege, pelo poder e pela flexibilidade, o tratamento de excessões estruturadas. Desta forma o C# também utiliza-se deste modelo estruturado, uniforme e *type-safe*.

Quando uma excessão ocorre, um objeto herdado de *System.Exception*, é criado para representá-la. O modelo orientado à objetos permite que seja criada um excessão definida pelo usuário que é herdada de *System.Exception* ou de uma outra classe de excessão pré-definida. As excessões pré-definas mais comuns são apresentadas na tabela 1.

As excessões podem ser disparadas de duas formas: através do comando *throw*, fornecendo a instância de uma classe herdada de *System.Exception*, ou em certas circunstâncias durante o processamento dos comandos e expressões que não podem ser completadas normalmente.

Os comando em C# para utilização do tratamento de excessões estruturados são: *try* – bloco de proteção do código, *catch* – filtra e trata a excessão, *finally* – sempre executado após o disparo da excessão ou não, e *throw* – dispara uma excessão.

Tabela 6: Classes de excessões mais comuns

Excessão	Descrição – Disparado quando
System.OutOfMemoryException	alocação de memória, através de *new*, falha.
System.StackOverflowException	quando a pilha(stack) está cheia e sobrecarregada.
System.NullReferenceException	uma referência nula(null) é utilizada indevidamente.
System.TypeInitializationException	um construtor estático dispara uma excessão.
System.InvalidCastException	uma conversão explícita falha em tempo de execução.
System.ArrayTypeMismatchException	o armazenamento dentro de um array falha.
System.IndexOutOfRangeException	o índice do array é menor que zero ou fora do limite.
System.MulticastNotSupportedException	a combinação de dois delegates não nulo falham.
System.ArithmeticException	DivideByZeroException e OverflowException. Base aritmética.
System.DivideByZeroException	ocorre uma divisão por zero.
System.OverflowException	ocorre um overflow numa operação aritmética. Checked.

Normalmente, o código abaixo é utilizado em alguns programas na verificação do sucesso da operação.

```
bool fSuccess = Process(100,200,300); //true se sucesso, false se falha
if(!fSuccess){
  System.Console.WriteLine("Processamento com erro");
  return;
}
```

Isto funciona corretamente, porém este não é o modelo usual e gera uma verificação para cada processamento da função, para confirmação do sucesso ou verificação de um erro. Em C#, o código acima será reescrito, qualquer erro será disparado através das excessões:

```
Process(100,200,300); //Gera excessão se falha
```

Comando *throw*

O comando *throw* é utilizado para disparar ou sinalizar a ocorrência de uma situação inesperada durante a execução do programa, ou seja uma excessão. O parâmetro seguido deve ser da classe System.Exception ou derivada.

```
using System;

class Throws{

  public static void Main(string[] args){
    //Verifica se somente uma string foi entrada
    if(args.Length==1)
      System.Console.WriteLine(args[0]);
    else{
      ArgumentOutOfRangeException ex;
      ex = new ArgumentOutOfRangeException("Utilize uma string somente");
      throw(ex); //Dispara a excessão
    }

  }

}
```

Outra forma para escrever o código acima, seria encurtar alguns passos:

```
using System;

class Throws{

  public static void Main(string[] args){
    //Verifica se somente uma string foi entrada
    if(args.Length==1)
      System.Console.WriteLine(args[0]);
    else
      //Dispara a excessão
      throw(new ArgumentOutOfRangeException("Utilize uma string somente"));

  }

}
```

Bloco *try - catch*

Uma ou mais instruções *catch* são colocadas logo abaixo do bloco *try* para interceptar uma excessão. Dentro do bloco *catch* é encontrado o código de tratamento da excessão. O tratamento da excessão trabalha de forma hierárquica, ou seja quando uma excessão é disparada, cada *catch* é verificado de acordo com a excessão e se a excessão ou derivada dela é encontrada o bloco será executado e os outros desprezados, por isso, na implementação é muito importante a sequência dos blocos *catch*. O *catch* também pode ser encontrado na sua forma isolada, tratando qualquer excessão não detalhada.

```
using System;
```

```
class Catchs{

  public static void Main(){

    int iMax=0;

    Console.Write("Entre um inteiro para valor máximo, entre 0 e o máximo será
sorteado:");

    try{

      iMax = Console.ReadLine().ToInt32();
      Random r = new Random();      //Instância a classe Random
      int iRand = r.Next(1,iMax);   //Sorteia randômincamente entre 0 e máximo
      Console.Write("O valor sorteado entre 1 e {1} é {0}", iRand, iMax);

    }
    catch(ArgumentException){
        Console.WriteLine("0 não é um valor válido");
    }
    catch(Exception e){
      Console.WriteLine(e);
    }

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc Catchs.cs*. Execute o programa digitando *Catchs*. A Figura 1, mostra compilação e execução da aplicação em C#. Note, na Figura 1, que primeira execução o valor 0 dispara a excessão *ArgumentException* que é tratatada e a terceira execução entra no caso mais genérico instânciando um objeto da classe *Exception*.

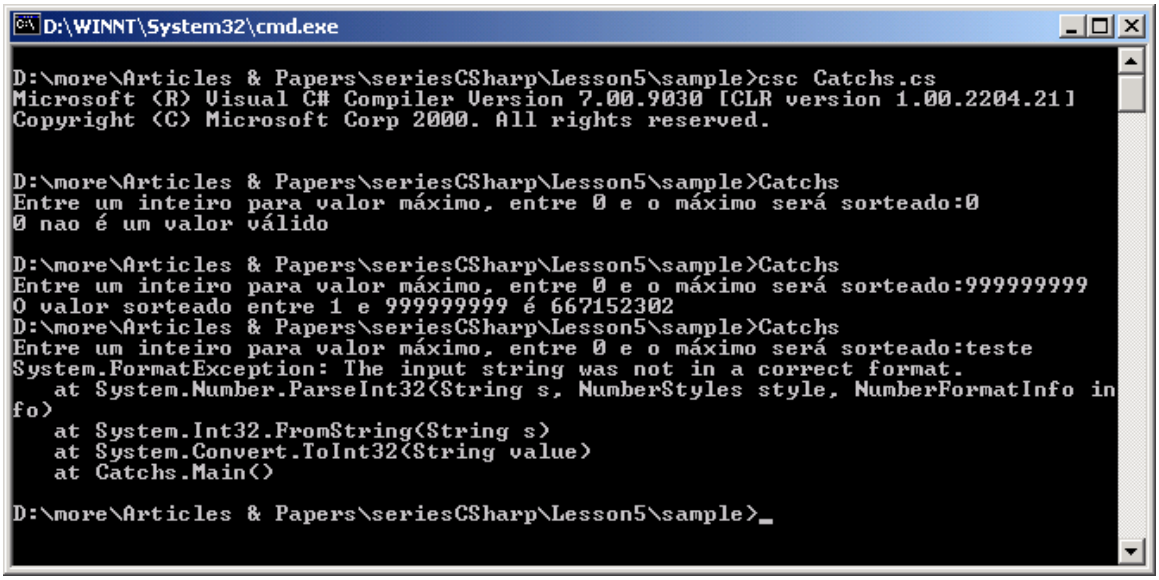

Figura 13: Compilação e Execução do exemplo Catchs

Outra possiblidade de tratamento de excessões é a forma mais genérica:

```
object o = (int)100;
try{
  string s = (string)o; //Produz a excessão InvalidCastException
}
catch{
  Console.Write("Ocorreu uma excessão");
}
```

Outra forma é reescrever o código acima, utilizando a classe *Exception* e obter uma informação mais detalhada sobre a excessão.

```
object o = (int)100;
try{
  string s = (string)o; //Produz a excessão InvalidCastException
}
catch(Exception e){
  Console.Write(e);
}
```

Bloco *try - finally*

A instrução *finally* garante a execução de seu bloco, independemente da excessão ocorrer no bloco *try*.

```
object o = (int)100;
try{
  OpenDB(); //Abre o banco de dados e produz a excessão
}
finally{
  CloseDB();
}
```

Tradicionalmente o bloco *finally* é utilizado para liberação de recursos consumidos, por exemplo fechar um arquivo ou uma conexão.

```
using System;

class TryFinally{

  public static void Main(){

    try{
      throw new Exception("A excessão..."); //Dispara a excessão
    }
    finally{
      Console.WriteLine("O bloco finally é sempre executado...");
    }

    Console.WriteLine("Esta linha não será executada...");

  }

}
```

Se não tratada, o comportamento de qualquer excessão é de terminação, como podemos concluir no exemplo acima. Lembrando que, o tratamento de uma excessão é, ou sua

interceptação é feita no bloco *catch*. E os comandos *try*, *catch* e *finally* podem ser utilizados em conjunto.

Bloco *try - catch - finally*

```
using System;
using System.Xml;

class TryCatchFinally{

public static void Main(){

  XmlDocument doc = null;

  try{

    doc = new XmlDocument();
    doc.LoadXml("<Exception>The Exception</Exception>"); //Carrega o conteúdo

    throw new Exception(doc.InnerText); //Dispara a excessão

  }
  catch(OutOfMemoryException){

    //Tratamento aqui

  }
  catch(NullReferenceException){

    //Tratamento aqui

  }
  catch(Exception e){

    //Tratamento aqui
    Console.WriteLine("Excessão ocorrida no programa {0}", e);

  }
  finally{
    Console.WriteLine(@"Gravando o Documento no C:\..."); //Uso do verbatim (@)
    doc.Save(@"c:\exception.xml"); //Grava o conteúdo
  }

  Console.WriteLine("Esta linha não será executada...");

}

}
```

Para compilar o exemplo acima, no prompt, digite *csc /r:System.Xml.dll TryCatchFinally.cs*. Execute-o digitando *TryCatchFinally*.

Fluxo e Rethrowing

O fluxo de execução do bloco *try* é extendido para suas chamadas entre funções. Sempre o bloco *try* mais interno tem prioridade, porém se o mesmo não existir a proteção passa para os blocos mais externos. Como no código abaixo, a função *Main* possui o bloco *try*, neste é

executado uma a função *Func1*, que chama a função *Func2* que tenta utilizar um índice inválido do vetor e isto gera a excessão *IndexOutOfRange* que é tratada no bloco *try* mais externo, o da função *Main*.

```
using System;

class Flux{

  static int[] arr = new int[3]; //Declara um array de inteiros de 3 posições

  public static void Main(){

   try{
      Func1();
    }
    catch(Exception e){
     Console.WriteLine(e);
    }

  }

  public static void Func1(){ Func2();  }
  public static void Func2(){ arr[3] = 100; }

}
```

Outra técnica que interfere no fluxo do tratamento das excessões é o *rethrowing*, ou seja a execução de um *throw* dentro do bloco *catch*, disparando um excessão que poderá ser tratado pelos blocos subsequentes. Verifique este comportamento no código abaixo:

```
using System;

class Rethrows{

  static int[] arr = new int[3]; //Declara um array de inteiros de 3 posições

  public static void Main(){

    try{
      Func1();
    }
    catch(Exception){ //Trata o rethrowing
      Console.WriteLine("O índice máximo para o array é {0}",
arr.GetUpperBound(0));
    }

  }

  public static void Func1(){
    try{
      Func2();
    }
    catch(IndexOutOfRangeException e){ //Trata a excessão gerada
     Console.WriteLine(e);
     throw e; //Rethrowing
    }

  }

  public static void Func2(){
```

```
    arr[3] = 100; //Excessão IndexOutOfRangeException
  }

}
```

Dentro de um bloco *try* é possível também encontrar outro bloco *try*. O código abaixo é válido, porém não é elegante, a técnica de *rethrowing* ou até mesmo uma boa construção de tratamentos de excessões poderiam ser utilizadas para este caso:

```
using System;

class TryTry{

  static int[] arr = new int[3];

  public static void Main(){

    try{
      Func1();
    }
    catch(IndexOutOfRangeException e){
      Console.WriteLine(e);
      try{
        throw e;
      }
      catch(Exception){
        Console.WriteLine("O índice máximo para o array é {0}",
arr.GetUpperBound(0));
      }

    }

  }

  public static void Func1(){ Func2(); }

  public static void Func2(){ arr[3] = 100; }

}
```

Tipos de tratamentos de excessões suportadas em C#

O C# suporta 3 tipos de tratamentos de excessões, são eles: *finally handlers*, *fault handlers* e *type-filtered handler*. Vimos suas aplicações anteriormente, agora vamos há definição:

Finally handlers são executados independemente se ocorrer excessão. Estes tratadores são representados pelo bloco *finally*.

```
try{
  throw new System.Exception();
}
finally{
  System.Console.Write(“Sempre sou executado”);
}
```

Fault handlers são executados sempre que uma excessão ocorre e interceptam qualquer tipo de erro. Estes tratadores são representados pelo bloco *catch* sem parâmetro, ou sem filtro.

```
try{
  throw new System.Exception();
}
catch{
  System.Console.Write(“Sou executado em caso de qualquer excessão”);
}
```

Type-filtered handler é executado quando uma excessão de um tipo específico ou sua base é gerada. Este tratador é representado pelo bloco *catch* com parâmetro, ou com filtro.

```
try{
  throw new System.MulticastNotSupportedException();
}
catch(System.NullReferenceException){
  System.Console.Write(“Sou executado em NullReferenceException”);
}
catch(System.ArithmeticException){
  System.Console.Write(“Sou executado em ArithmeticException”);
}
catch(System. MulticastNotSupportedException){
  System.Console.Write(“Sou executado em MulticastNotSupportedException”);
}
```

Outro tipo de tratador, permitido pelo .NET, é o *user-filetered handler* que pode ser simulado em C#, conforme o código abaixo:

```
using System;

class UserFilter{

  public static void Main(){

    int iDivd=0, iDivs=0;

    Console.Write("Entre um inteiro para o Dividendo:");
    iDivd = Console.ReadLine().ToInt32();
    Console.Write("Entre um inteiro para o Divisor:");
    iDivs = Console.ReadLine().ToInt32();

    try{

      double dResult = (double)iDivd/iDivs;
      Console.WriteLine("{0}/{1} = {2,0:N5}",iDivd,iDivs,dResult);

    }
    catch(Exception){

      //Simulando o filtro
      if(iDivd==0 && iDivs==0){
        Console.WriteLine("0/0 é indeterminado");
        return;
      }

      if(iDivs==0){
        Console.WriteLine("0 como divisor é indefinido");
        return;
      }

    }
```

```
  }

}
```

A classe Exception

A forma mais comum e generalizada de disparo de uma excessão é através da classe base *Exception*. Ela fornece as informações necessárias para tratamento das excessões, possuindo alguns membros, métodos e propriedades, que trazem as informações necessárias decorrentes do erro. Normalmente uma instância de classe, ou derivada, é utilizada através de um bloco *catch*, como vimos nos exemplos anteriores.

Vamos descreever alguns membros da classe *Exception*.

Message retorna uma string com o texto da mensagem de erro.

```
catch(System.Exception e){
  System.Console.WriteLine(e.Message);
}
```

Source possui ou define a uma string com o texto da origem(aplicação ou objeto) do erro.

```
catch(System.Exception e){
  System.Console.WriteLine(e.Source);
}
```

HelpLink possui uma string com o link(URN ou URL) para arquivo de ajuda.

```
catch(System.Exception e){
  System.Console.WriteLine(e.HelpLink);
}
```

StackTrace possui uma string com a sequência de chamadas na *stack*. Quando não ocorrer uma excessão a *call stack* pode ser obtida por *Environment.StackTrace*.

```
catch(System.Exception e){
  System.Console.WriteLine(e.StackTrace);
}
```

InnerException retorna uma referência para uma excessão interna.

```
throw e.InnerException;
```

TargetSite retorna o método que disparou esta excessão, este retorno será tratado através da classe *MethodBase* encontrada no namespace *System.Reflection*.

```
System.Reflection.MethodBase mb = e.TargetSite;
if(mb.IsStatic) Console.Write("Membro que disparou a excessão é static");
```

GetBaseException retorna uma referência para uma excessão interna. Internamente este método utiliza a propriedade *InnerException*.

```
throw e.GetBaseException();
```

SetHelpLink define o link(URN ou URL) para arquivo de ajuda. Este método fornece o valor para a propriedade *HelpLink*.

```
e.SetHelpLink("http://www.microsoft.com/brasil/msdn");
```

```
using System;

public class Exceptions{

  public static void Main(){

  try{
    FillStack();
  }
  catch(Exception e){

    //Utilizando os membros da classe Exception
    Console.WriteLine("Exception Members");
    e.Source = "internal FillStack Function";
    Console.WriteLine("Source: {0}",e.Source);
    Console.WriteLine("Message: {0}",e.Message);
    e.SetHelpLink(@"C:\Microsoft.Net\FrameworkSDK\Docs\cpref.chm");
    Console.WriteLine("HelpLink: {0}",e.HelpLink);
    Console.WriteLine("StackTrace: {0}",e.StackTrace);
    System.Reflection.MethodBase mb = e.TargetSite;
    if(mb.IsStatic) Console.Write("Membro que disparou a excessão é static");

  }

  }

  internal static void FillStack(){
    //Simulando esta linha - FillStack();
    throw new StackOverflowException();
  }

}
```

Os principais construtores da classe são:

- *public Exception();* - inicialização padrão.
- *public Exception(string);* - inicialização com uma mensagem de erro específica.
- *public Exception(string, Exception);* - inicialização com uma mensagem de erro específica e referência a uma excessão interna.

A classe *Exception* suporta a interface *ISerializable* que permite o objeto controlar a serialização e deserialização.

Classe de excessão definida pelo usuário

Basicamente criar um nova classe de excessão é herdar da classe *Exception* ou derivadas, sobrecarregar e criar alguns membros. Abaixo a classe de excessão *MyOwnException* é

criada e seus construtores são sobrecarregados, porém o valor passado para ele é transmitido para sua classe pai através da palavra-chave *base*.

```
using System;
using System.WinForms;

internal class MyOwnException: Exception{

  public MyOwnException():base(){}
  public MyOwnException(string msg):base(msg){}
  public MyOwnException(string msg, Exception ie):base(msg,ie){}

}

public class MyApp{

  public static void Main(){

    try{
      throw new MyOwnException("Excessão disparada...",new CoreException());
    }
    catch(MyOwnException e){
      e.Source = "MyApp class";
      MessageBox.Show(e.Message,e.Source,MessageBox.OK|MessageBox.IconError);
    }

  }

}
```

Para compilar o exemplo acima, no prompt, digite *csc /r:System.WinForms.dll MyOwnException.cs*. Conforme Figura 2, execute-o digitando *MyOwnException*.

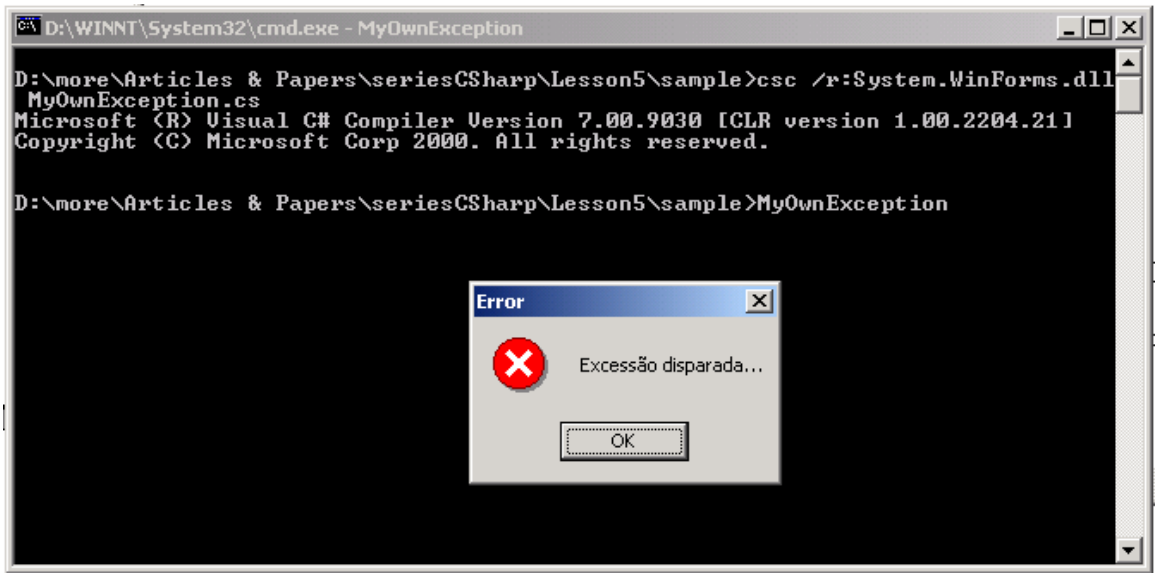

Figura 14: Compilação e Execução do exemplo MyOwnException

Conclusão

Neste artigo, é apresentado o tratamento de excessão estruturado e seus poderosos recursos para proteção do código contra possíveis anômalias. Vale lembrar que seu uso é restrito para condições de erro, não sendo indicado para situações onde eventos podem ser empregados, como por exemplo chegar no final de um arquivo. Outro ponto importante é que ao contrário de outras linguagens, a utilização do tratamento de excessão não é um processo custozo. O tratamento de excessão, gera um pequeno overhead, porém este é gerenciado pelo garbage collector. Utilizar este mecanismo tráz muitos benefícios para aplicação final.

Links

http://msdn.microsoft.com/library/welcome/dsmsdn/drguinet04242001.htm
http://msdn.microsoft.com/vstudio/nextgen/default.asp
http://www.microsoft.com/net/default.asp
http://msdn.microsoft.com/net/
http://gotdotnet.com/
http://www.microsoft.com/brasil/msdn