Apostila C#

Conceitos Básicos

Universidade Tecnológica Federal do Paraná

Diogo Cezar Teixera Batista

Cornélio Procópio

20 de novembro de 2008

Apostila baseada nas obras de : Fabio R. Galuppo, Edwin Lima, Breno Gontijo e Rudson Kiyoshi

Sumário

Lista de Tabelas

Lista de Códigos

1 INTRODUÇÃO

Entende-se por uma plataforma de desenvolvimento, um conjunto de instruções ordenadas que tem por objetivo resolver um problema do mundo real, abstraindo-o em um conjunto de comandos lógicos.

1.1 Plataforma .NET

.NET é a nova plataforma de desenvolvimento da Microsoft que tem como foco principal o desenvolvimento de Serviços *WEB* XML. Um serviço *Web* XML, ou simplesmente *Web Service* transcende ao que nós conhecemos como páginas dinâmicas, as quais podem ser acessadas a partir de um *browser*. A idéia central de um *Web Service* consiste em permitir que as aplicações, sejam elas da *Web* ou *Desktop*, se comuniquem e troquem dados de forma simples e transparente, independente do sistema operacional ou da linguagem de programação.

1.1.1 Abordagem .NET

- *Independência de linguagem de programação*: o que permite a implementação do mecanismo de herança, controle de exceções e depuração entre linguagens de programação diferentes;
- *Reutilização de código legado*: o que implica em reaproveitamento de código escrito usando outras tecnologias como COM, COM+, ATL, DLLs e outras bibliotecas existentes;
- *Tempo de execução compartilhado*: o "*runtime*" de .NET é compartilhado entre as diversas linguagens que a suportam, o que quer dizer que não existe um *runtime* diferente para cada linguagem que implementa .NET;
- *Sistemas auto-explicativos e controle de versões*: cada peça de código .NET contém em si mesma a informação necessária e suficiente de forma que o runtime não precise procurar no registro do Windows mais informações sobre o programa que está sendo executado. O runtime encontra essas informações no próprio sistema em questão e sabe qual a versão a ser executada, sem acusar aqueles velhos conflitos de incompatibilidade ao registrar DLLs no Windows;
- Simplicidade na resolução de problemas complexos.

1.1.2 Arquitetura .NET

1.1.2.1 CLR (*Commom Language Runtime*) O CLR, ou tempo de execução compartilhado, é o ambiente de execução das aplicações .NET. As aplicações .NET não são aplicações Win32 propriamente ditas (apesar de executarem no ambiente Windows), razão pela qual o runtime Win32 não sabe como executá-las. O Win32, ao identificar uma aplicação .NET, dispara o runtime .NET que, a partir desse momento, assume o controle da aplicação no sentido mais amplo da palavra, porque, dentre outras coisas, é ele quem vai cuidar do gerenciamento da memória via um mecanismo de gerenciamento de memória chamado *Garbage Collector* (GC) ou coletor de lixo. Esse gerenciamento da memória torna os programas menos susceptíveis a erros. Mais ainda, o CLR como seu próprio nome o diz, é compartilhado e, portanto, não temos um runtime para VB.NET, outro para C# etc. É o mesmo para todo mundo.

1.1.2.2 CLS (*Common Language Specification*) O CLS, ou EspecificaçãoComumda Linguagem, é um subconjunto do CTS, e define um conjunto de regras que qualquer linguagem que implemente a .NET 8 deve seguir a fim de que o código gerado resultante da compilação de qualquer peça de software escrita na referida linguagem seja perfeitamente entendido pelo runtime .NET.

1.1.2.3 BCL (*Base Classe Library*) Oferece ao desenvolvedor uma biblioteca consistente de componentes de software reutilizáveis que não apenas facilitem, mas também que acelerem o desenvolvimento de sistemas.

2 A linguagem C#

C# (pronunciada "C Sharp"), é uma linguagem de programação da Plataforma .NET, derivada de C/C++ orientada à objetos. É a linguagem nativa para .NET *Common Language Runtime*(CLR), mecanismo de execução da plataforma .NET. Isso possibilita a convivência com várias outras linguagens especifícadas pela Common Language Subset(CLS). Por exemplo, uma classe base pode ser escrita em C#, derivada em Visual Basic e novamente derivada em C#.

2.1 Características do C#

- *Simplicidade*: os projetistas de C# costumam dizer que essa linguagem é tão poderosa quanto o C++ e tão simples quanto o Visual Basic.

- *Completamente orientada a objetos*: em C#, qualquer variável tem de fazer parte de uma classe.
- *Fortemente tipada*: isso ajudará a evitar erros por manipulação imprópria de tipos, atribuições incorretas etc.
- *Gera código gerenciado*: assim como o ambiente .NET é gerenciado, assim também é a linguagem C#.
- *Tudo é um objeto*: System.Object é a classe base de todo o sistema de tipos de C#.
- *Controle de versões*: cada assembly gerado, seja como EXE ou DLL, tem informação sobre a versão do código, permitindo a coexistência de dois assemblies homônimos, mas de versões diferentes no mesmo ambiente.
- *Suporte a código legado*: o C# pode interagir com código legado de objetos COM e DLLs escritas em uma linguagem não-gerenciada.
- *Flexibilidade*: se o desenvolvedor precisar usar ponteiros, o C# permite, mas ao custo de desenvolver código não-gerenciado, chamado "unsafe".
- *Linguagem gerenciada*: os programas desenvolvidos em C# executam num ambiente gerenciado, o que significa que todo o gerenciamento de memória é feito pelo runtime via o GC (*Garbage Collector*), e não diretamente pelo programador, reduzindo as chances de cometer erros comuns a linguagens de programação onde o gerenciamento da memória é feito diretamente pelo programador.

3 PRIMEIRO PROGRAMA

Escrevendo o tradicional programa Hello World, em C#:

Código 1: Hello World em C#

```
1 using System;
2 class Hello{
3         public static void Main(){
4                 Console.WriteLine("Hello World!!!");
5         }
6 }
```

A cláusula *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*.

3.1 Main

O método *Main* é o ponto de entrada de execução do seu programa. A classe que será executada inicialmente possui embutida a função estática *Main*. Uma ou mais classes podem conter a função *Main*, entretanto, apenas uma será o ponto de entrada, indicada na compilação pelo parâmetro `/main:<tipo>`, ou simplificando `/m:<tipo>`.

O método Main, pode ser declarado de 4 formas:

1. Retornando um vazio(void): `public static void Main()`;
2. Retornando um inteiro(int): `public static int Main()`;
3. Recebendo argumentos, através de um array de string e retornando um vazio: `public static void Main(string[] args)`;
4. Recebendo argumentos, através de um array de string e retornando um inteiro: `public static int Main(string[] args)`.

4 ESTRUTURA DE UM PROGRAMA

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

Código 2: Estrutura de um programa em C#

```
1  using System;
2  namespace MathNamespace{
3          public class MathClass{
4          /* Main: exibe no prompt */
5                  public static void Main(string[] args){
6                          Math m = new Math();
7                          Console.WriteLine(m.Sum(1,1));
8                  }
9                  //<summary>Classe Math</summary>
10                 public class Math:Object{
11                         public int Sum(int a, int b){
12                                 return (a+b);
13                         }
14                 }
15         }
16 }
```

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

- ***Namespaces***: são a forma lógica de organizar o código-fonte;
- **Tipos**: classes, estruturas, interfaces, delegações, enums;
- **Membros**: constantes, campos, métodos, propriedades, indexadores, eventos, operadores, construtores;
- **Outros**: coméntarios e instruções.

5 VARIÁVEIS NA LINGUAGEM C#

Na linguagem C# as variáveis estão agrupadas em algumas categorias como:

- *Static*: existe apenas uma única copia desta variável para todas as instâncias de uma classe. Uma variável *static* começa a existir quando um programa começa a executar, e deixa de existir quando o programa terminar.
- *Instance*: existe uma copia para cada instância de uma classe. Uma variável Instance começa a existir quando uma instância daquele tipo e criado, e deixa de existir quando não houver nenhuma referencia aquela instância ou quando e método *Finalize* e executado.
- *Array*: é uma matriz que e criada em tempo de execução.

5.1 Tipos de dados

Como toda linguagem de programação o C# também 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, parâmetros, declarações e até mesmo em comparações.

A Tabela 1 apresenta os tipos básicos da linguagem C# relacionados juntamente com os tipos de dados do .NET 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 x 10^{-28} a 7,9 x 10^{28}
double	System.Double	Flutuante IEEE 64-bit com	+-5,0 x 10^{-324} a +-1,7 x 10^{324}
float	System.Single	Flutuante IEEE 32-bit com	+-1,5 x 10^{-45} a +-3,4 x 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

5.1.1 Tipos Valor e Tipos Referência

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

- Tipos valor(*value types*);
- Tipos referência(*reference types*);
- Tipos ponteiro(*pointer types*).

Tipos valor armazenam dados em memória enquanto tipos referência armazenam uma referência, ou o endereço, para o valor atual.

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.

Os tipos ponteiro, apenas apontam para um endereço de memória.

5.1.2 Conversão de Tipos

Converter um tipo de dado em número ou em literal é comum em situações de programação. Devemos considerar alguns aspectos para a conversão de números:

- Como existem diversos tipos de números, inteiros, ponto flutuante ou decimal, os valores são convertidos sempre para o tipo de maior faixa de valores. Por exemplo, o tipo long é convertido para o ponto flutuante, mais é importante ressaltar que o contrario causa um erro.
- Os tipos de menor faixa são convertidos para os de maior faixa. Por exemplo, o tipo int pode ser convertido para: long, float, double ou decimal.
- A conversão dos tipos de ponto flutuante(float, double) para decimal causa erro.
- A conversão entre os tipos com sinal e sem sinal de valores inteiros com o mesmo tamanho causa erro. Por exemplo, entre o tipo int e unit.

Por exemplo:

Código 3: Exemplo de conversão automática de tipos

```
1 int VarInteiro = 32450;
```

```
2 long VarLong = VarInteiro;
3 float VarFloat = VarLong;
4 double VarDouble = VarFloat;
5 decimal VarDecimal = VarLong;
6 byte VarByte = (byte)VarInteiro;
7 int VarInteiro = (int)31.245F;
```

Tabela 2: Tipos de conversão automática

Tipo	Converte em
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	long, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double

5.1.3 O Objeto *Convert*

Em C# temos o objeto ***Convert*** que é usado para converter um tipo de dado em outro.

Os tipos de dados suportados são: *Boolean*, *Char*, *SByte*, *Byte*, *Int16*, *Int32*, *Int64*, *UInt16*, *UInt32*, *UInt64*, *Single*, *Double*, *Decimal*, *DateTime* e *String*.

Por exemplo:

Código 4: Exemplo de utilização do objeto ***Convert***

```
1 double dNumber = 23.15;
2
3 int    iNumber = System.Convert.ToInt32(dNumber);
4 bool   bNumber = System.Convert.ToBoolean(dNumber);
5 String strNumber = System.Convert.ToString(dNumber);
6 char   chrNumber = System.Convert.ToChar(strNumber[0]);
```

5.2 *Arrays*

Um tipo *array* é uma matriz de valores do mesmo tipo, que é criada em tempo de execução, podendo ser acessada por meio de um índice.

A declaração do *array* sempre faz o uso de um colchete([ ]) depois do tipo da variável. O uso da instrução *new* sempre deve ser utilizado, pois é obrigatório.

O tipo *array* pode ter diversas dimensões, o tamanho desta é definido pelo desenvolvedor, mas devemos saber que o primeiro índice é sempre zero.

No tipo *array* devemos sempre inicializar seus elementos, pois é obrigatório também. Veja abaixo a forma de sintaxe para a declaração de arrays.

Código 5: Sintaxe para a declaração de *Arrays*

```
1 <TIPO>[ ] NomeDoArray = new <TIPO> [ tamanho do array ];
2
3 float[] ValorIndice = new float [10];
4 string[] ElementoVetor = new string[10];
```

Código 6: Sintaxe para a declaração de *Arrays* com duas ou mais dimensões

```
1 <TIPO> [,] NomeDoArray = new <TIPO> [tamanho do array,tamanho do array];
2
3 float [,] ValorIndice = new float [10,10];
4 string [,,] ElementoVetor = new string [10,10,10];
```

Código 7: Sintaxe para a declaração de uma matriz de *Arrays* com duas ou mais dimensões

```
1 <TIPO> [][] NomeDoArray = new <TIPO> [tamanho do array] [tamanho do array];
2
3 float [][] ValorIndice = new float [10][10];
4 string [][][] ElementoVetor = new string[10][10][10];
```

Código 8: Sintaxe para a inicialização de *Arrays* com duas ou mais dimensões

```
1 <TIPO> [] NomeDoArray = new <TIPO> [tamanho do array]{valores separados por ,};
2
3 float [] ValorIndice = new float[5]{1.25,2,3.23,1.32,5};
```

```
4
5  string [,] ElementoVetor = new string[3,3] {{"ab", "ac", "bc"} ,
6  {"ab", "ac", "bc"}};
7
8  int [][] MatrizDeInteiro = new int [2][];
9  MatrizDeInteiro[ 0 ] = new int [ 5 ] {1,3,5,7,9};
10 MatrizDeInteiro[ 1 ] = new int [ 4 ] {2,4,6,8};
```

Para passar um argumento array para um método, especifique o nome do array sem usar colchetes e para que um método receba um *array*, a lista de parâmetros deve especificar que um array será recebido.

Veja o seguinte exemplo:

Código 9: Passando *Arrays* à métodos

```
1  int[] vetor = {10, 20, 30, 40, 50};
2  Array.IndexOf(vetor, 22);
3
4  public void ExibeVetor( int[] vetor );
```

Algumas Propriedades e Métodos dos Arrays:

- `obj.Length` ⟶ Tamanho do vetor;
- `Array.IndexOf(Array vetor, object value)` ⟶ Procura a primeira ocorrência de valor em vetor;
- `Array.LastIndexOf(Array vetor, object value)` ⟶ Procura a última ocorrência de valor em vetor;
- `Array.Sort(Array vetor)` ⟶ Ordena um vetor crescentemente;
- `Array.Reverse(Array vetor)` ⟶ Ordena um vetor decrescentemente.

6 COMANDOS

6.1 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.

6.1.1 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.

Código 10: Exemplo do comando `if` em C#

```
1  //if com uma única possibilidade. Exibe a string "Verdadeiro" no Console caso a
2  //expressão (a==true) seja verdadeira
3  if(a==true){
4  System.Console.Write("Verdadeiro");
5  }
6  //if com uma única possibilidade. Exibe a string "Verdadeiro" no Console caso a
7  //expressão (a==true) seja verdadeira, senão exibe a string "Falso"
8  if(a==true){
9    System.Console.Write("Verdadeiro");
10 }
11 else{
12   System.Console.Write("Falso");
13 }
```

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:

Código 11: *Ifs* com *And* e *Or*

```
1  //&& (And). Somente a primeira função é executada
2  if(MyFunc() && MyFunc2());
3
4  //|| (Or). Ambas funções são executadas
5  if(MyFunc() || MyFunc2());
6
7  public static bool MyFunc(){ return false; }
8
9  public static bool MyFunc2(){ return true; }
```

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

Código 12: *Ifs* aninhados

```
1 if(x==1){
2   if(y==100){
3     if(z==1000){
4       System.Console.Write("OK");
5     }
6   }
7 }
```

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

Código 13: Curto-circuito

```
1 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.

Código 14: Exemplo *if-else-if*

```
1  using System;
2  class Ifs{
3    public static void Main(){
4      char chOpt;
5      Console.WriteLine("1-Inserir");
6      Console.WriteLine("2-Atualizar");
7      Console.WriteLine("3-Apagar");
8      Console.WriteLine("4-Procurar");
9      Console.Write("Escolha entre [1] a [4]:");
10     //Verifica se os valores entrados esta entre 1 e 4
11     //caso contrário pede reentrada
12     do{
13       chOpt = (char)Console.Read();
14     }while(chOpt<'1' || chOpt>'4');
15     if(chOpt=='1'){
```

```
16        Console.WriteLine("Inserir...");
17        //InsertFunction();
18      }
19      else if(chOpt=='2'){
20        Console.WriteLine("Atualizar...");
21        //UpdateFunction();
22      }
23      else if(chOpt=='3'){
24        Console.WriteLine("Apagar...");
25        //DeleteFunction();
26      }
27      else{
28        Console.WriteLine("Procurar...");
29        //FindFunction();
30      }
31  }
```

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.

Código 15: Operador Ternário

```
1  int x;
2  if(f==true)
3    x = 100;
4  else
5    x = 1000;
6
7  // As linhas acima podem ser substituídas por:
8
9  int x = f==true?100:1000;
```

6.1.2 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*:

Código 16: Comando *switch*

```
1  using System;
2  class Switchs{
3    public static void Main(){
4      char chOpt;
5      Console.WriteLine("1-Inserir");
6      Console.WriteLine("2-Atualizar");
7      Console.WriteLine("3-Apagar");
8      Console.WriteLine("4-Procurar");
9      Console.Write("Escolha entre [1] a [4]:");
10     //Verifica se os valores entrados esta entre 1 e 4
11     //caso contrário pede reentrada
12     do{
13       chOpt = (char)Console.Read();
14     }while(chOpt<'1' || chOpt>'4');
15     switch(chOpt){
16     case '1':
17       Console.WriteLine("Inserir...");
18       //InsertFunction();
19       break;
20     case '2':
21       Console.WriteLine("Atualizar...");
22       //UpdateFunction();
23       break;
24     case '3':
25       Console.WriteLine("Apagar...");
26       //DeleteFunction();
27       break;
28     default:
29       Console.WriteLine("Procurar...");
30       //FindFunction();
31     }
32 }
```

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:

Código 17: Comando *switch*

```
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");
}
```

A cláusula *break*, é utilizada para separar os blocos do *switch* e garante que o bloco seja executado somente até determinado ponto.

6.2 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*.

6.2.1 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.

Código 18: Iteração *for*

```
for(int x=0; x < 100; ++x){
        System.Console.WriteLine(x);
}
for(;;){ // Laço infinito
        System.Console.WriteLine("Hello, World!");
}
```

```
for(int y=100, int x = 0;x < y; ++x, --y){ // Laço com mais de 1 variável
        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.

Código 19: Iteração *for* (exemplo)

```
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
    for(int a=0; a < iVezes; ++a){
      Console.Write(iSeq[a]);
```

```
23        Console.Write(" ");
24      }
25    }
26  }
```

6.2.2 Comando *foreach*

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

Código 20: Iteração *foreach* (exemplo)

```
1  using System;
2  class Fibonacci{
3    public static void Main(){
4      int iVezes;
5      Console.Write("Entre de 1 a 100 para o n° de elementos a exibir na sequência
           de Fibonacci:");
6      //Verifica se os valores entrados esta entre 1 e 100
7      //caso contrário pede reentrada
8      do{
9        iVezes = Console.ReadLine().ToInt32();
10     }while(iVezes<1 || iVezes>100);
11     //Cria o vetor dinâmicamente
12     int[] iSeq = new int[iVezes];
13     iSeq[0] = 1;
14     //Preenche o vetor
15     if(iVezes > 1){
16       iSeq[1] = 1;
17       for(int a=2; a < iVezes; ++a)
18       iSeq[a] = iSeq[a-1] + iSeq[a-2];
19     }
20     //Exibe o vetor
21     foreach(int a in iSeq){
22       Console.Write(a);
23       Console.Write(" ");
24     }
25   }
26 }
```

6.2.3 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.

Código 21: Iteração *do while* (exemplo)

```
1  // Comando while:
2  int a = 0;
3  bool f = true;
4  while(f){
5
6    if(++a==100) f = true;
7    System.Console.WriteLine(a);
8  }
9  // Comando do ... while:
10 int a = 0;
11 bool f = true;
12 do{
13   if(++a==100) f = true;
14   System.Console.WriteLine(a);
15 } while(f);
```

7 OPERADORES

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 3 apresenta essas divisões.

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 parênteses.

Tabela 3: Operadores do C#

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

7.1 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.

Os operadores unários (atribuídos a 1 atributo) + e - são utilizados para representar se o número é positivo ou negativo, respectivamente.

Código 22: Operadores Unários

```
1 x = +1000 // x = 1000
2 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:

Código 23: Operadores Binários

```
1 string x = "Hello" + "World"          // x = "HelloWorld"
2 string x = "Valor = " + 100           // x = "Valor = 100"
3 int    x = 1000 % 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:

Código 24: Exemplo Operadores Binários

```
1  using System;
2  class Arithmetics{
3    public static void Main(string[] args){
4      //Verifica o número de argumentos entrados
5      if(args.Length == 3){
6        int x=0,y=0;
7        //Convertem os valores dos argumentos 2 e 3 para inteiro 32-bit
8        //Se ocorrer algum erro o modo de utilização
9        try{
10         x = args[1].ToInt32();
11         y = args[2].ToInt32();
12       }
13       catch{
14         usage();
15         return;
16       }
17       //Efetua a operação selecionada no primeiro argumento
18       switch(args[0]){
19       case "+":
20         Console.Write("Valor da soma = {0}", x+y);
21         break;
```

```
22        case "-":
23          Console.Write("Valor da subtração = {0}", x-y);
24          break;
25        case "/":
26          Console.Write("Valor da divisão = {0}", x/y);
27          break;
28        case "*":
29          Console.Write("Valor da multiplicação = {0}", x*y);
30          break;
31        case "%":
32          Console.Write("Valor da sobra = {0}", x%y);
33          break;
34        default:
35          usage();
36    }
37    else{
38      usage();
39    }
40  }
41
42  public static void usage(){
43    //Modo de utilização
44    Console.WriteLine("Modo de usar: Arithmetics operador valor1 valor2");
45    Console.WriteLine("Ex.: Arithmetics + 100 200");
46  }
47  }
```

7.2 Operadores de 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.

Código 25: Operadores de Incremento e Decremento

```
1  int x = 1000; // x = 1000
2  x++;          // x = 1001
3  int y = x++;  // x = 1002 , y = 1001
4  x--;          // x = 1001
5  y = --x;      // x = 1000 , y = 1000
```

```
6  ++x;            // x = 1001
7  --x;            // x = 1000
8  y = ++x;        // x = 1001 , y = 1001
```

7.3 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.

Código 26: Exemplo do operador de negação

```
1  bool x = true;
2  bool y = !x // y = false;
3  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.

7.4 Operação 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.

Código 27: Exemplo do operador de atribuição

```
1  int x = 10;
```

```
2 int y = x;
3 int z = x + y;
```

Os operadores +=, -=, *=, /=, %=, &=, —=, ≙, ¡¡= e ¿¿= representam a atribuição composta, que normalmente atuam como um atalho na construção de uma expressão.

Código 28: Exemplo do operador de atribuição composta

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

8 PROGRAMAÇÃO BASEADA EM OBJETOS

8.1 Convenções e Padrões de Nomenclatura

As convenções e padrões de nomenclatura auxiliam o desenvolvedor na organização, localização e manutenção de seus códigos. Imagine como seria encontrar a seguinte codificação:

Código 29: Exemplo de codificação sem qualquer padrão de nomenclatura

```
1 using System;
2 class classeparacadastrodeclientes{
3   private int j;
4   private int n;
5   private string str;
6   public adicionarcliente(string s){
7   }
8 }
```

Observe a diferença, se utilizarmos algumas regras de nomenclatura:

Código 30: Exemplo de codificação com padrão de nomenclatura

```
1 using System;
2 class ClasseCadastrodeClientes{
3   private int numFilhos;
4   private int numBens;
5   private string nomeCompleto;
6   public adicionarCliente(string cliente){
```

```
7     }
8 }
```

Após aplicadas as regras de nomenclaturas, além de saber o que significa cada variável e método, conseguimos ler sem dificuldade as palavras sem os espaços que as separam.

Para a nomenclatura de classes, como visto no exemplo, deve-se utilizar o seguinte padrão: Primeiro caractere de cada palavra em caixa alta. Chamada de notação *PascalCasing*.

Para a nomenclatura de métodos, utiliza-se: Primeira letra de cada palavra em caixa alta, menos da primeira. Chamada de notação *camelCasing*.

8.1.1 Nomeação de variáveis

Recomendação da Microsoft para nomeação de variáveis:

- Evite usar underline "_";
- Não crie variáveis com o mesmo nome mudando somente entre maiúsculas e minúsculas;
- Utilize nomes de variáveis com minúsculas;
- Evite utilizar todas as letras maiúsculas (normalmente utilizado para definir constantes);
- Notação *camelCasing*.

8.1.2 Nomeação de classes, métodos, propriedades, entre outros.

Recomendações para nomeação de classes, métodos, propriedades, enumeradores, interfaces, constantes, campos somente leitura e namespace: Notação *PascalCasting*.

8.2 Classes

Uma classe é um poderoso tipo de dado em C#. Como estrutura, uma classe define os dados e o comportamento dos tipos de dados.

Uma classe em C# tem a seguinte estrutura:

Código 31: Exemplo de Classe em C#

```
1 class NomeDaClasse {
2         // Definição dos atributos
3         private int atrib1;
```

```
4           private string atrib2;
5           // Método construtor
6           public NomeDaClasse(int param1, string param2){
7           }
8           // Definição dos métodos
9           public tipoRetorno MetodoUm([lista de parâmetros]){
10          return [valor];
11          }
12  }
```

Os atributos definem através de tipos de dados as características que um objeto venha a apresentar. Em uma classe Carro, por exemplo, poderíamos afirmar que cor, peso e modelo seriam seus atributos.

O método construtor sempre é chamado quando se cria um novo objeto, ele pode ou não conter uma assinatura.

Os outros métodos definem através de rotinas as ações que um objeto venha a apresentar, No exemplo citado acima, uma classe Carro poderia conter os métodos: acelerar, frear e ligar.

8.3 Propriedades

As propriedades são recursos fornecidos pelas classes para que seja possível alterar seus valores.

Os recursos *get* e *set* comuns às propriedades.

Veja como é possível se definir esses recursos:

Código 32: Exemplo de Propriedades em C#

```
1   public tipodedado NomeDaPropriedade {
2           get{
3                   return nomeAtributo;
4           }
5           set{
6                   nomeAtributo = value;
7           }
8   }
9   // Utiliza-se da seguinte maneira
10  this.NomeDaPropriedade = valor;
11  valor = this.NomeDaPropriedade;
```

8.4 Modificadores de visibilidade

C# apresenta os seguintes modificadores de visibilidades:

1. *private*: Significa que, exceto a classe incluída, nada pode acessar o objeto, método ou variável;
2. *public*: Significa que todos têm acesso livre a este membro;
3. *protected*: Significa que são apenas visíveis para as classes derivadas por meio de herança;
4. *internal*: todos têm acesso livre a este membro dentro de um assembly (DLL ou EXE; correspondente ao JAR do Java). Fora do assembly a classe é inacessível.

Atributos *internal*, são utilizados geralmente para a criação de bibliotecas, já que uma biblioteca pode ter vários *namespaces*.

O exemplo a seguir mostra como devemos dispor os modificadores de visibilidade:

Código 33: Exemplo de utilização dos modificadores de visibilidade C#

```
1 class NomeDaClasse {
2         private   int atrib1;
3         public    int atrib2;
4         protected int atrib3;
5         internal  int atrib4;
6         ...
7 }
```

8.5 Herança

A herança é um recurso utilizado para derivar classes que têm métodos ou atributos em comum. Sua principal vantagem é o reaproveitamento de código.

Para ilustrar a utilização do recurso de herança, imagine a seguinte situação:

[**?**]m um determinado programa, deve-se implementar as seguintes classes: Fornecedor(id, cnpf, rua, numero, bairro, telefone, email, gerente) e Cliente(id, cpf, rua, numero, bairro, telefone, email, limiteDeCompras). O programa deve cadastrar essas informações, bem como consulta-las e imprimi-las.

Se o programador optar por utilizar herança, os atributos e métodos em comum:

- Atributos:
  - rua;
  - numero;
  - bairro;
  - telefone;
  - email.
- Metodos:
  - cadastrar();
  - consultar();
  - imprimir().

Poderiam ser escritos em uma "Pai" chamada Pessoa que derivaria as duas classes filhas Fornecedor e Cliente.

Em C# todas as classes derivam da classe *Object*.

Sua declaração deve acontecer da seguinte maneira:

Código 34: Exemplo de declaração de herança em C#

```
1 class NomeDaClasse : ClasseBase {
2 ...
3 }
```

8.5.1 *This* e *Base*

As cláusulas *this* e *base* são referências que indicam a propria classe e a classe base, respectivamente.

Entende-se como classe base, a classe cuja a classe atual herda as propriedades e atributos.

Sua notação pode ser observada no seguinte exemplo:

Código 35: Exemplo de *this* e *base* em C#

```
1 this.nomeAtributo = valor;
2 valor = this.nomeAtributo;
3 this.NomeMetodo();
```

```
4
5 base.nomeAtributoClasseBase = valor;
6 valor = base.nomeAtributoClasseBase;
7 base.NomeMetodoClasseBase();
```

8.6 Declaração e Chamada de Métodos e Objetos

Para se instanciar um objeto deve-se utilizar a operação *new*. Essa operação atribui um <objeto> montado dentro de uma variável do tipo <objeto>.

Utiliza-se a seguinte notação:

Código 36: Exemplo instanciação de objeto em C#

```
1 MinhaClasse obj = new MinhaClasse();
```

Para acessar seus atributos e métodos utilizamos a instrução ".", como pode ser observado no exemplo a seguir:

Código 37: Exemplo acesso a atributos e métodos em C#

```
1 obj.nomeMetodo();
2 obj.nomeAtributo = 23;
3 obj.NomePropriedade = "Apenas um teste";
```

Entretanto como já foi explanado, atributos com modificador de acesso do tipo *private* não possibilitam o acesso direto, para isso devemos utilizar as propriedades explanadas na sessão 8.3.

8.7 Métodos e Atributos *Static*

A operação *static* define um método ou atributo como pertencentes à classe em questão e não aos objetos, assim sendo esses atributos ou métodos terão apenas 1 cópia para n objetos que gerarem.

Sua declaração é feita com a palavra *static* depois do modificador de acesso (*public, private*) e antes do tipo de dado (*int, string*).

O seu acesso é feito pelo nome da classe e não mais pela referência da classe ou pelo nome do objeto.

Veja no exemplo:

Código 38: Exemplo acesso a atributos e métodos estáticos em C#

```
1 NomeDaClasse.atributoEstatico = valor;
2 valor = NomeDaClasse.atributoEstatico;
3 NomeDaClasse.MetodoEstatico();
```

8.8 *Const* e *ReadOnly*

São operadores utilizados para a criação de constantes, cujos os valores não poderão ser alterados durante a execução do programa.

Algumas diferenças entre os operadores:

- *const*:
  - Não pode ser estático (*static*);
  - O valor é setado em tempo de compilação;
  - É inicializado somente na compilação.
- *readonly*:
  - Pode ser estático (*static*);
  - O valor é setado em tempo de execução;
  - Pode ser inicializado na declaração ou na codificação do construtor.

8.9 Classes e Métodos Abstratos

A classe abstrata é um tipo de classe que somente pode ser herdada e não instanciada, de certa forma pode se dizer que este tipo de classe é uma classe conceitual que pode definir funcionalidades para que as suas subclasses (classes que herdam desta classe) possam implementa-las de forma não obrigatória, ou seja ao se definir um conjunto de métodos na classe abstrata não é de total obrigatoriedade a implementação de todos os métodos em suas subclasses, em uma classe abstrata os métodos declarados podem ser abstratos ou não, e suas implementações devem ser obrigatórias na subclasse ou não, quando criamos um método abstrato em uma classe abstrata sua implementação é obrigatória, caso você não implemente o mesmo o compilador criará um erro em tempo de compilação.

O mesmo se têm com métodos abstratos, não se pode herda-los.

Código 39: Exemplo de implementação de uma classe abstrata em C#

```
1  abstract class formaClasse
2  {
3      abstract public int Area();
4  }
5  class quadrado : formaClasse
6  {
7      int x, y;
8      // Se não for implementado o método Area()
9      // será gerado um compile-time error.
10     // Utiliza-se o operador override para indicar a sobrescrita.
11     public override int Area()
12     {
13         return x * y;
14     }
15 }
```

8.10 Interfaces

As interfaces são fundamentais em um sistema orientado a objetos, quando dizemos que um objeto é a instancia de uma classe, na verdade queremos dizer, que este objeto implementa a interface definida pela classe, ou seja uma interface define as operações que um objeto será obrigado a implementar. Para cada operação declarada por um objeto deve ser especificado o nome da operação, os objetos que esta operação aceita como parâmetro e o tipo de valor retornado pela operação; este conjunto de informações sobre uma determinada operação tem o nome de assinatura da operação, e um conjunto de assinaturas de operações dá-se o nome de interface.

É importante lembrar que uma interface nunca contém implementação, ou seja numa interface não se pode definir campos, pois o mesmo é uma implementação de um atributo objeto, a interface também não permite construtores pois num contrutor temos as instruções usadas para inicializar campos. Para podermos usar uma interface devemos criar uma classe ou estrutura e herdar da interface, com isso é obrigatório implementar todos os métodos da interface.

Código 40: Exemplo de implementação de uma interface em C#

```
1  interface IExemploInterface
2  {
```

```
3       void ExemploMetodo();
4   }
5   class ImplementacaoClasse : IExemploInterface
6   {
7       // Implementação explicita da interface
8       void IExemploInterface.ExemploMetodo()
9       {
10          // Implementação do método
11      }
12      static void Main()
13      {
14          // Declarando uma instancia de uma interface
15          IExemploInterface obj = new ImplementacaoClasse();
16          // chame o método.
17          obj.ExemploMetodo();
18      }
19  }
```

Note que, para se sobrescrever um método da interface utilizamos <*Interface*>.<*Metodo*>. Código 40, Linha 8.

A declaração de uma instância de uma interface é feita de forma diferente da declaração de um objeto normal, aqui temos: Interface <var> = new <ClasseQueImplementaAInterface>(); Código 40, Linha 15.

8.11 Métodos Virtuais

Quando queremos possibilitar a algum método que ele seja sobrescrito, utilizamos o operador *virtual*. Os métodos virtuais podem possuir corpo.

Caso um método não seja declarado como *virtual* ou *abstract* não será possível a sua sobrescrita.

8.12 Classes e Métodos *sealed* - Finais

Uma classe selada é utilizada para restringir características da herança do objeto, quando uma classe é definida como sealed , está classe não poderá ser herdada, caso você tente o compilador criara um erro em tempo de compilação, após criar uma classe selada pode se observar que o intelisense (famoso ctrl + espaço) não mostra o nome da classe definida como sealed quando você tenta criar uma herança para novas classes.

Os métodos declarados como *sealed* também não poderão ser sobrescritos.

Código 41: Exemplo de implementação de uma classe *sealed* em C#

```
1  sealed class ClasseSelada {
2      public int x;
3      public int y;
4  }
5  class MainClass {
6      static void Main()     {
7          ClasseSelada sc = new ClasseSelada();
8          sc.x = 110;
9          sc.y = 150;
10         Console.WriteLine("x = {0}, y = {1}", sc.x, sc.y);
11     }
12 }
```

8.13 Então, quando devo utilizar o que?

Classes Abstratas podem adicionar mais funcionalidades, sem destruir as funcionalidades das classes filhos que poderiam estar usando uma versão mais antiga. Elas fornecem uma maneira simples e fácil para versionar nossos componentes. Através da atualização da classe base, todas as classes que herdam são atualizadas automaticamente com a mudança.

Em uma interface, a criação de funções adicionais terá um efeito sobre suas classes filhos, devido à necessidade de implementação dos Métodos criados na interface.

Classes abstratas deveriam ser usadas principalmente para objetos que estão estritamente relacionados, enquanto o uso de interfaces é mais adequado para fornecer funcionalidade comum a classes independentes. Digamos que existem duas classes, de passaros e de aviões, e nas duas existam os métodos chamados voar(). Seria estranho para uma classe aviões herdar a partir de umas classe passaros apenas porque necessita do método voar(). Em vez disso, o método voar() deve ser definido em uma interface e em ambas as classes passaros e aviões devem implementar a interface. Se queremos proporcionar uma funcionalidade em comum para os componentes, devemos utilizar uma classe abstrata.

Classes abstratas nos permite implementar parcialmente uma classe, enquanto a interface não contem a implementação de qualquer membro. Por isso, a seleção de interface ou classes abstratas depende das necessidades e design do nosso projeto. Podemos fazer uma classe

abstrata, interface, ou até uma combinação de ambas dependendo de nossas necessidades. Se desejarmos criar uma classe ou método interno para um componente ou library o ideal é utilizar o tipo *sealed* porque qualquer tentativa de anular algumas das suas funcionalidades não será permitida.

Nós podemos marcar uma classe ou método como selados por motivos comerciais, a fim de impedir um terceiro de modificar nossa classe. Por exemplo, no .NET a string é uma classe selada. Não devemos usar a palavra-chave *sealed* com um método a menos que o método seja uma mudança de outro método, ou se estamos definindo um novo método e não queremos que ninguém mais sobreponha-o, não se deve declará-lo como virtual em primeiro lugar. A palavra-chave selado fornece uma maneira de garantir que ao sobrepor um método seja fornecido um "final"significa que ninguém mais poderá sobrepor-lo novamente.

9 TRATAMENTO DE ERROS E EXCEÇÕES

No mundo dos *frameworks* e linguagens de programação, as excessões, ações que causam anômalias nas aplicações são tratadas de diversas formas. 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 uma 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 9.

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.

9.1 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.

Tabela 4: Classes de excessões mais comuns em C#

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.

Código 42: Exemplo de utilização do comando throw

```
1  using System;
2  class Throws{
3    public static void Main(string[] args){
4      //Verifica se somente uma string foi entrada
5      if(args.Length==1)
6        System.Console.WriteLine(args[0]);
7      else{
8        ArgumentOutOfRangeException ex;
9        ex = new ArgumentOutOfRangeException("Utilize uma string somente");
10       throw(ex); //Dispara a excessão
11     }
12   }
13 }
```

9.2 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.

Código 43: Exemplo de utilização do bloco *try - catch*

```
1  using System;
2  class Catchs{
3    public static void Main(){
4      int iMax=0;
5      Console.Write("Entre um inteiro para valor máximo, entre 0 e o máximo será
           sorteado:");
6      try{
7        iMax = Console.ReadLine().ToInt32();
8        Random r = new Random();       //Instância a classe Random
9        int iRand = r.Next(1,iMax);    //Sorteia randômincamente entre 0 e máximo
10       Console.Write("0 valor sorteado entre 1 e {1} é {0}", iRand, iMax);
11     }
12     catch(ArgumentException){
13         Console.WriteLine("0 não é um valor válido");
14     }
15     catch(Exception e){
16       Console.WriteLine(e);
17     }
18   }
19 }
```

9.3 Bloco *try - finally*

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

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

Código 44: Exemplo de utilização do bloco *try - finally*

```
1  using System;
2  class TryFinally{
3    public static void Main(){
4      try{
5        throw new Exception("A excessão..."); //Dispara a excessão
6      }
7      finally{
8        Console.WriteLine("O bloco finally é sempre executado...");
9      }
10     Console.WriteLine("Esta linha não será executada...");
11   }
12 }
```

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

9.4 Bloco *try - catch - finally*

Os comandos try, catch e finally podem ser utilizados em conjunto, como pode ser visto no proximo exemplo:

Código 45: Exemplo de utilização do bloco *try - catch - finally*

```
1  using System;
2  using System.Xml;
3  class TryCatchFinally{
4  public static void Main(){
5    XmlDocument doc = null;
6    try{
7      doc = new XmlDocument();
8      doc.LoadXml("<Exception>The Exception</Exception>"); //Carrega o conteúdo
9      throw new Exception(doc.InnerText); //Dispara a excessão
10   }
11   catch(OutOfMemoryException){
12     //Tratamento aqui
13   }
```

```
14    catch(NullReferenceException){
15      //Tratamento aqui
16    }
17    catch(Exception e){
18      //Tratamento aqui
19      Console.WriteLine("Excessão ocorrida no programa {0}", e);
20    }
21    finally{
22      Console.WriteLine(@"Gravando o Documento no C:\...."); //Uso do verbatim (@)
23      doc.Save(@"c:\exception.xml"); //Grava o conteúdo
24    }
25    Console.WriteLine("Esta linha não será executada...");
26  }
27  }
```

9.5 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 descrever alguns membros da classe *Exception*:

- *Message*: retorna uma string com o texto da mensagem de erro.
- *Source*: possui ou define a uma string com o texto da origem(aplicação ou objeto) do erro.
- *HelpLink*: possui uma string com o link(URN ou URL) para arquivo de ajuda.
- *StackTrace*: possui uma string com a sequência de chamadas na stack.
- *InnerException*: retorna uma referência para uma excessão interna.
- *TargetSite*: retorna o método que disparou esta excessão.
- *GetBaseException*: retorna uma referência para uma excessão interna.
- *SetHelpLink*: define o link(URN ou URL) para arquivo de ajuda.

Código 46: Membros da classe *Exception*

```
1  catch(System.Exception e){
2    System.Console.WriteLine(e.Message);
3  }
4
5  catch(System.Exception e){
6    System.Console.WriteLine(e.Source);
7  }
8
9  catch(System.Exception e){
10   System.Console.WriteLine(e.HelpLink);
11 }
12
13 catch(System.Exception e){
14   System.Console.WriteLine(e.StackTrace);
15 }
16
17 throw e.InnerException;
18
19 System.Reflection.MethodBase mb = e.TargetSite;
20 if(mb.IsStatic) Console.Write("Membro que disparou a excessão é static");
21
22 throw e.GetBaseException();
23
24 e.SetHelpLink("http://www.microsoft.com/brasil/msdn");
```

10 MANIPULAÇÃO DE ARQUIVOS

Com a manipulação de arquivos, podemos criar, editar e excluir arquivos ou diretórios.

Para que possamos trabalhar com as classes referente a manipulação de arquivos devemos utilizar o pacote: `System.IO` (IO *Input/Output* significa Entrada/Saída).

As principais classes que estão nesse pacote:

10.1 Classes *DirectoryInfo* e *FileInfo*

É possível recuperar algumas informações de arquivos e diretórios, como data de criação, extensão, entre outros. Para isso utiliza-se em C# as classes *DirectoryInfo* e *FileInfo*.

Utiliza-se *DirectoryInfo* quando se deseja informações sobre um diretório, e *FileInfo* para

Tabela 5: Principais classes do System.IO

Classe	Uso
Directory, File, DirectoryInfo, e FileInfo	Cria, exclui e move arquivos e diretórios. Ainda retorna informações específicas sobre arquivos ou diretórios
FileStream	Usado para escrever/ler informações em arquivo com ajuda das classes StreamReader e StreamWriter
StreamWriter e StreamReader	Lê e escreve um informação textual
StringReader e StringWriter	Lê e escreve um informação textual a partir de um *buffer* de string

informações de um arquivo.

A seguir estão listadas algumas propriedades e métodos que essas classes oferecem:

Tabela 6: Propriedades e métodos de *DirectoryInfo* e *FileInfo*

Propriedade/Método	Uso
Attributes	Retorna os atributos associados aos arquivos
CreationTime	Retorna a hora de criação do arquivo
Exists	Checa se o arquivo/diretório existe
Extension	Retorna a extensão do arquivo
LastAccessTime	Retorna a hora do último acesso
FullName	Retorna o nome completo do arquivo/diretório
LastWriteTime	Retorna a hora da ultima escrita no arquivo/diretório
Name	Retorna o nome do arquivo/diretório
Delete()	Exclui o arquivo/diretório

10.1.1 Criando diretórios e subdiretórios

Para criar um diretório utiliza-se a seguinte notação:

Código 47: Criação de diretório

```
1 DirectoryInfo dir1 = new DirectoryInfo(@"F:\WINNT");
```

Para criar um subdiretório:

Código 48: Criação de subdiretórios

```
1 DirectoryInfo dir = new DirectoryInfo(@"F:\WINNT");
2 try{
3   dir.CreateSubdirectory("Sub");
4   dir.CreateSubdirectory(@"Sub\MySub");
5 }
6 catch(IOException e){
7   Console.WriteLine(e.Message);
8 }
```

10.1.2 Acessando as propriedades

Para acessar as propriedades de um diretório utiliza-se a seguinte notação:

Código 49: Propriedades de um diretório

```
1 Console.WriteLine("Full Name is : {0}", dir1.FullName);
2 Console.WriteLine("Attributes are : {0}", dir1.Attributes.ToString());
```

Abaixo um exemplo de acesso às propriedades de arquivos:

Código 50: Propriedades de arquivos

```
1  DirectoryInfo dir = new DirectoryInfo(@"F:\WINNT");
2  FileInfo[] bmpfiles = dir.GetFiles("*.bmp");
3  Console.WriteLine("Total number of bmp files", bmpfiles.Length);
4  Foreach( FileInfo f in bmpfiles)
5  {
6    Console.WriteLine("Name is : {0}", f.Name);
7    Console.WriteLine("Length of the file is : {0}", f.Length);
8    Console.WriteLine("Creation time is : {0}", f.CreationTime);
9    Console.WriteLine("Attributes of the file are : {0}",
10                     f.Attributes.ToString());
11 }
```

10.2 Criando arquivos usando a classe *FileInfo*

Com a classe *FileInfo*, é possível criar novos arquivos, acessar suas informações, exclui-los e move-los. Essa classe também oferece métodos para abrir, ler e escrever um arquivo.

O seguinte exemplo mostra como é possível criar um arquivo texto e acessar suas informações.

Código 51: Criando arquivos com a classe FileInfo

```
1  FileInfo fi = new FileInfo(@"F:\Myprogram.txt");
2  FileStream fstr = fi.Create();
3  Console.WriteLine("Creation Time: {0}",fi.CreationTime);
4  Console.WriteLine("Full Name: {0}",fi.FullName);
5  Console.WriteLine("FileAttributes: {0}",fi.Attributes.ToString());
6  //Way to delete Myprogram.txt file.
7  Console.WriteLine("Press any key to delete the file");
8  Console.Read();
9  fstr.Close();
10 fi.Delete();
```

10.2.1 Entendendo o método *Open()*

Com o método *Open()*, disponível na classe *FileInfo*, é possível abrir um arquivo. Deve-se passar no construtor, o modo de abertura e acesso ao arquivo.

O seguinte exemplo ilustra essa situação:

Código 52: Abrindo arquivos com a classe FileInfo

```
1  FileInfo f = new FileInfo("c:\myfile.txt");
2  FileStream s = f.Open(FileMode.OpenorWrite, FileAccess.Read);
```

10.2.2 Entendendo a classe *FileStream*

Ao abrir ou criar nossos arquivos, o atribuímos para a classe *FileStream*. Ela pode escrever ou ler arquivos, com a ajuda das classes *StreamWriter* e *StreamReader*.

O exemplo a seguir ilustra como isso é possível:

Código 53: Escrevendo/Lendo com FileStream

```
1  FileStream fs = new FileStream(@"c:\mcb.txt", FileMode.OpenOrCreate, FileAccess.
       Write);
2  StreamWriter sw = new StreamWriter(fs);
3  sw.write("teste");
4  sw.writeLine("teste");
5  sw.Close();
6
7  FileStream fs = new FileStream(@"c:\mcb.txt", FileMode.OpenOrCreate, FileAccess.
       Write);
8  StreamReader sr = new StreamReader(fs);
9  string texto;
10 texto = sr.ReadToEnd();
11 sr.Close();
```

10.2.3 Métodos *CreateText()* e *OpenText()*

O método *CreateText()* retorna um *StreamWriter* que vai escrever um arquivo. O método *OpenText()* retorna um *StreamReader* que vai ler um arquivo.

Esses métodos são utilizados quando trabalha-se com arquivos de texto puro.

Exemplos de utilização dos métodos:

Código 54: CreateText e OpenText

```
1  FileInfo fi = new FileInfo("c:\myfile.txt");
2  StreamReader txtR;
3  txtR = fi.OpenText();
4  string read = null;
5  while ((read = txtR.ReadLine()) != null){
6    Console.WriteLine(read);
7  }
8  s.Close();
9  // Método ReadToEnd();
10 Console.WriteLine(txtR.ReadToEnd());
11
12 FileInfo fi = new FileInfo("c:\myfile.txt");
13 StreamWriter txtW;
14 txtW = fi.CreateText();
15 txtW.Write("teste");
16 txtW.Close();
```

11 APROFUNDANDO EM WINDOWS FORMS

Um programa dificilmente é desenvolvido com apenas um formulário (*Form*). Sempre temos vários deles nos nossos programas, vamos estudar neste capitulo como trabalhar com varios formulários. Além disso, vamos estudar as propriedades mais importantes dos formulários.

Basicamente podemos ter os seguintes tipos de interface Windows:

- MDI (*Multiple Document Interface*): Aplicação que suporta múltiplos documentos abertos simultaneamente, como o Word por exemplo.
- SDI (*Single Document Interface*): Aplicação que permite a abertura de apenas um documento de cada vez. Exemplo: O Paint do Windows, a calculadora, o Internet Explorer.
- Janelas modais. Exemplo: As janelas informativas conhecidas como diálogos.

11.1 Aplicações MDI

As aplicações MDI têm como objetivo criar várias instâncias (filhos) dentro de uma aplicação principal (pai). Por esse motivo temos que seguir alguns passos para configurar esse ambiente.

Para definir uma janela como principal (janela pai) altera-se a propriedade `isMdiContainer = true` ainda em tempo de compilação.

11.1.1 Adicionando uma nova janela filha

Para adicionar uma nova janela filha, devemos inserir um novo formulário ao projeto: menu *File/Add > New Item*, selecione *Windows Forms*.

Ao instanciarmos uma nova janela filha utilizamos a seguinte codificação:

Código 55: Janela filha (MDI)

```
1 wndFilha myWnd = new wndFilha( );
2 myWnd.MdiParent = this;
3 myWnd.Text = "Janela Filha";
4 myWnd.Show();
```

O que se fez aqui?

- Criamos uma nova instância do formulário que constitui a nossa janela filha;

- Em seguida, definimos o pai da janela filha ajustando a propriedade `MdiParent` do formulário como sendo o formulário principal. Por essa razão, atribuímos a essa propriedade o valor this, indicando que a janela pai é o objeto correspondente ao formulário principal.
- Finalmente mostramos a janela filha chamando o método *Show()*.

11.1.2 Fechando uma janela filha

Para fechar uma janela filha MDI ativa utiliza-se a seguinte notação:

Código 56: Fechar janela filha (MDI)

```
1 if (this.MdiChildren.Length != 0)
2 this.ActiveMdiChild.Close( );
```

11.2 Aplicações SDI

Sempre que você cria uma Windows Application um formulário já é criado por padrão, ele é conhecido como formulário Base.

Para adicionar um novo formulário no seu programa você:

1. Na janela *Solution Explorer*, clique com o botão direito no nome do projeto, selecione *Add* e clique em *Windows Form*;
2. Digite um nome para o novo formulário e clique em *Add*.

11.2.1 Exemplo múltiplos formulários

Vamos fazer um exemplo que ilustra como podemos trabalhar com múltiplos formulário:

1. Crie uma Windows Application chamada Formulários;
2. Arraste 2 *buttons* para o Form1;
3. Mude as propriedades *Text* dos *Buttons* 1 e 2 para Vermelho e Verde respectivamente;
4. Adicione mais dois formulários no projeto;
5. Mude o nome do *Form2* para frmVermelho e do Form3 para frmVerde. Para isso clique sobre o nome do *Form* no *Solution Explorer* e clique em *Rename*;

- Se a uma mensagem de confirmação aparecer, clique em Sim. Isso vai atualizar todas as referencias ao *form* do projeto para o novo nome.

6. Mude a propriedade *BackColor* do frmVermelho para a cor Vermelha;
7. Mude a propriedade *BackColor* do frmVerde para Verde;
8. No Form1 de um clique duplo sobre o botão Vermelho e digite o seguinte código dentro do procedimento do evento *click*:

Código 57: Código para exibir formulário

```
1 frmVermelho Vermelho = new frmVermelho();
2 Vermelho.Show();
```

9. Faça o mesmo para o botão Verde mas digite o seguinte código:

Código 58: Código para exibir formulário com show dialog

```
1 frmVerde Verde = new frmVerde();
2 Verde.ShowDialog();
```

10. Execute a aplicação.

Note que para a chamada de um formulário é necessário que o mesmo seja instanciado.

Com o formulário Vermelho aberto, perceba que você pode clicar novamente no Form1 e depois voltar para o Vermelho e assim por diante, o que não acontece com o formulário Verde. Isso porque usamos o método ShowDialog ao invés do método Show.

11.3 Passando valores entre Forms

Continuando com o exemplo anterior, vamos alterar um componente de um formulário, em outro:

1. Adicione 1 *button* e 1 *label* no formulário frmVermelho;
2. Vá para o painel de código do frmVermelho e dentro da classe frmVermelho digite o seguinte código:

Código 59: Variável pública do tipo string

```
1 public string mensagem;
```

Isso cria uma variável publica do tipo string chamada Mensagem.

3. Vá para o Form1 e de um clique duplo sobre o botão Vermelho.

4. Adicione o seguinte código:

Código 60: Modificando o valor da string

```
1 Vermelho.mensagem = "Formulario Form1";
```

5. Volte para o frmVermelho, de um clique duplo sobre o button1 e digite o seguinte código no procedimento:

Código 61: Modificando o valor da label para um string local

```
1 label1.Text = mensagem;
```

Isso adiciona o conteúdo da variável mensagem no Label1.

6. Execute a aplicação.

O conteúdo da variável foi exibida no label1. O mais importante desta lição é que você agora é capaz de passar valores entre formulários.

11.4 Posicionando os formulários na tela

Por questões de estética, temos a possibilidade de alterar algumas propriedades dos componentes para que eles se tornem mais apresentáveis. O posicionamento de formulário na tela ilustra uma dessas possibilidades.

Vamos estudar algumas modificações na propriedade *StartPosition*.

- *CenterScreen*: Essa propriedade faz com que o formulário em questão seja exibido no centro da tela;

- *Manual*: Deve-se especificar manualmente como os valores que serão usados para posicionar o formulário na tela.

  – A propriedade location faz o posicionamento x, y do formulário na tela. Se alterarmos a propriedade para (100, 50), o formulário será exibido a 100 pixels da lateral esquerda e 50 pixels do topo da tela.

A propriedade *WindowsState* altera o estado inicial do meu formulário principal:

- *Normal*: Abre o formulário em seu estado inicial;
- *Minimized*: Abre o formulário minimizado na barra de tarefas;
- *Maximized*: Abre o formulário maximizado.

11.5 Controlando os eventos dos formulários

Os eventos são ações atribuídas ao comportamento do formulário. Sempre que um evento ocorre um bloco de código pode ser processado, esse bloco de código é conhecido como *Manipulador de Evento*.

O *.NET Framework* usa uma nomeação padrão para os *Manipuladores de Eventos*. Essa nomeação combina o nome do objeto com o evento correspondente ligando-os por um underline, exemplo: button1_Click; form1_Load.

12 CONEXÃO COM BANCO DE DADOS

12.1 O que é o ADO.NET ?

O ADO.NET é uma nova tecnologia baseada no ADO (*Active Data Objects*), com muito mais recursos. O ADO.NET possui um modelo para manipulação de dados completamente diferente da versão anterior do ADO, simplificando o processo de conexão e manipulação de dados.

A arquitetura do ADO.NET foi criada para trabalhar com um ambiente desconectado, ou seja, buscamos as informações do banco de dados e trazemos para a memória da aplicação. A manipulação dos dados é feita toda em memória e posteriormente enviada ao banco de dados.

Por trabalhar de uma forma voltada ao modelo desconectado, o ADO.NET possui uma camada de persistência em XML. É possível gravar e ler todo o conteúdo de todo um conjunto de armazenado nas estruturas do ADO.NET em XML.

O ADO.NET faz parte do .NET Framework, e é composto por um conjunto de classes, interfaces, tipos e enumerações.

12.2 Os *namespaces* relacionados ao ADO.NET

Para trabalharmos com o ADO.NET em uma aplicação .NET, é necessário utilizarmos algumas das *namespaces* disponíveis nas bibliotecas do .NET Framework. Alguns dos principais *namespace* são:

- *System.Data*: Contém a infra-estrutura básica para trabalharmos com qualquer base de dados relacional. Neste *namespace* encontramos as classes responsáveis por armazenar as estruturas dos bancos relacionais em memória;
- *System.Data.Common*: Contém as interfaces comuns a todos os bancos de dados. Este *namespace* é utilizado internamente pelo framework e por fabricantes de bancos de dados, para a construção de bibliotecas de acesso;
- *System.Data.SqlClient*: Biblioteca de acesso ao SQL Server. Permite a conexão, a extração e a execução de comandos em servidores SQL Server de versão 7 ou superior;
- *System.Data.OleDb*: Biblioteca de acesso para bancos de dados que suportam OleDb. Permite conexão, a extração e a execução de comandos nestes bancos de dados. É necessário informar o provedor *OleDb* a ser utilizado. Permite acesso a bancos mais simples, como o Access;

- *System.Data.SqlTypes*: Contém a definição dos tipos nativos do SQL Server;
- *System.XML*: Contém as classes para manipulação de documentos XML. Como o ADO.NET possui uma camada de persistência em XML, este *namespace* é amplamente utilizado.

12.3 O modelo de execução do ADO.NET

O ADO.NET provê uma estrutura de acesso a dados que permite o acesso a múltiplas bases de dados simultaneamente. É possível armazenar duas tabelas de diferentes bancos de dados (SQL Server e Access por exemplo) em uma mesma estrutura de dados (DataSet).

A estrutura responsável pelo armazenamento dos dados é o *DataSet*, que contém um conjunto de objetos (*DataTables*) que representam resultados tabulares extraídos da base de dados.

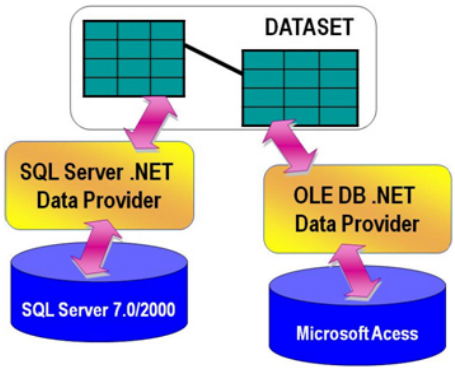

Figura 1: Esquema acesso ao banco de dados

Para fazer a extração dos dados, o ADO.NET utiliza os chamados ".NET Data Providers". Os *Data Providers* são bibliotecas de classes especializadas para o acesso a um tipo de banco de dados relacional. Por serem uma implementação específica para o banco de dados, estas possuem um acesso bem mais eficiente do que outras bibliotecas como a OleDb.

Segue uma lista de data providers disponíveis para os bancos citados:

- Firebird .NET Data Provider;
- MySQL .NET Data Provider;
- Npgsql .NET Data Provider;

Apesar de serem uma implementação específica para um tipo de banco de dados, as classes dos *Data Providers* possuem uma estrutura comum.

12.4 O modelo de execução em um ambiente conectado

O ADO.NET é capaz de trabalhar com dois modelos, o modelo conectado e o modelo desconectado. No modelo conectado é necessário manter a conexão aberta enquanto são realizadas as operações de leitura e gravação.

Para trabalharmos em um modelo conectado, devemos observar alguns objetos disponíveis nas classes dos .NET *Data Providers*, que devem ser utilizados na seguinte ordem:

- *XxxConnection*: É o objeto utilizado para estabelecer a conexão com o banco. É necessário informar os parâmetros de conexão e abrir a conexão com o banco. Exemplos dessa classe são *SqlConnection* e *OleDbConnection*;
- *XxxCommand*: É um objeto utilizado para enviar comandos a base de dados. É necessário montar a cláusula Sql desejada e informar ao objeto de comando. Ao executar o comando, este pode devolver um objeto do tipo *XxxDataReader*;
- *XxxDataReader*: É um objeto utilizado para ler dados de um comando executado. O método Read lê os dados de registro em registro. Após a leitura completa dos dados é necessário fechar o *DataReader* e a conexão;

12.5 O modelo de execução em um ambiente desconectado

O modelo de execução em um ambiente desconectado utiliza outros objetos. Neste modelo utilizamos o objeto *DataSet* para armazenar e manipular os dados em memória e o objeto *XxxDataAdapter* para extrair e enviar as alterações ao banco de dados. O objeto de conexão também é utilizado neste modelo.

Os passos para extração e manipulação dos dados em um ambiente desconectado são:

1. É aberta uma conexão utilizando um objeto *XxxConnection* (*OleDbConnection* ou *SqlConnection* por exemplo);
2. É criado um objeto do tipo *XxxDataAdapter* que é responsável por fazer a extração de dados do banco de dados para a memória e o posterior envio dos dados da memória para o banco de dados;
3. Utilizando o método *Fill*, extraímos os dados da base de dados e armazenamos em um *DataSet*. Neste momento fechamos a conexão com o banco pois os dados já estão na memória da aplicação para serem manipulados;

4. Como os dados estão em memória, é possível inserir, remover ou alterar registros do *DataSet*;

5. Ao finalizar as alterações, restabelecemos a conexão com o banco de dados para enviar as alterações;

6. Utilizando o método *Update* do *DataAdapter*, enviamos as alterações para o banco de dados. O *DataAdapter* verifica os tipos de alterações que foram realizadas e executa o comando correspondente no banco de dados (inserção, exclusão, atualização);

7. Ao finalizar o processo, fechamos a conexão com o banco de dados.

12.6 Estabelecendo uma conexão com um banco de dados

O primeiro passo para criarmos uma aplicação com conexão a um banco de dados é estabelecer a conexão com o banco. Para estabelecermos a conexão, devemos criar um objeto de conexão.

Ao criarmos uma instância da classe que irá se conectar, devemos informar uma string de conexão, que contém todos os parâmetros para a conexão com o banco de dados, como usuário e senha.

A string de conexão possui uma série de parâmetros, que pode variar de acordo com o banco de dados utilizado. Os parâmetros da string de conexão são separados por ponto e vírgula, e devem ser informados com a seguinte notação:

Código 62: Padrão para *Connection Strings*

```
1 "Nome do Parâmetro = Valor do Parâmetro"
```

Abaixo está uma classe com string de conexão para os bancos MySQL, Firebird e PostGreSQL respectivamente:

Código 63: Padrão para *Connection Strings*

```
1 class Global
2 {
3         public static string cnmysql = "database = dados; data source =
              localhost; user id = root; password = password";
4
```

```
5           public static string cnfirebird = "User=SYSDBA; Password = masterkey;
                Database = C:\\dados.fdb; DataSource = localhost; Dialect=3; Charset
                =WIN1252; Role=; Connection lifetime=15; Pooling=true; MinPoolSize
                =0; MaxPoolSize=50; Packet Size=4096; ServerType=0";
6
7           public static string cnpostgree = "User ID=postgres;Password=password;
                Host=localhost;Port=5432;Database=dados;Pooling=true;Min Pool Size
                =0;Max Pool Size=100;Connection Lifetime=0";
8   }
```

12.7 Criando comandos

É possível executar comando no banco de dados através da classe *SqlCommand*. Ao criar um objeto dessa classe, devemos informar o comando Sql a ser executado, bem como a conexão a ser utilizada. Estes parâmetros podem ser informados tanto no construtor da classe *SqlCommand* como também através das propriedades *CommandText* e *Connection*.

Os comandos SQL informados em um objeto de comando podem ser de qualquer tipo: que retornam um conjunto de linha, que retornam um valor específico, ou que não retornam nenhuma quer. Cada um destes tipos de comando SQL possui um método para execução.

Através da classe *SqlCommand* também é possível executar *Stored Procedures* do banco de dados, sendo necessário apenas informar o nome da *stored procedure* no parâmetro *CommandText*, e setar a propriedade *CommandType* da classe para *CommandTypes.StoredProcedure*.

Um exemplo de criação de um objeto de comando pode ser observado a seguir.

Código 64: Exemplo de utilização do comando *SqlCommand*

```
1 SqlCommand oCmd = New SqlCommand("UPDATE Products SET UnitPrice=UnitPrice*1.1");
2 oCmd.Connection = oConn;
3 oCmd.CommandText = "UPDATE Products SET UnitPrice=UnitPrice*1.05";
```

12.8 Executando comandos

Para executarmos os comandos especificados na classe *SqlCommand*, precisamos executar um dos métodos de execução disponíveis. Os métodos de execução variam de acordo com a natureza do comando executado. Os três métodos mais comuns são:

- *ExecuteNonQuery*: Para comandos que não executam consultas (querys);

- *ExecuteScalar*: Para comandos que executam resultados escalares;
- *ExecuteReader*: Para comandos que retornam conjuntos de dados.

12.8.1 O método *ExecuteNonQuery*

O método *ExecuteNonQuery* é utilizado quando queremos executar um comando que não retorna como resultado um conjunto de dados. Este método é utilizado para comandos de atualização e inserção e também para comando DCL (*Data Control Language*) suportados pelo banco de dados.

Opcionalmente podemos informar um parâmetro para este método para obter o número de linhas afetadas pelo comando executado.

Um exemplo de utilização do comando *ExecuteNonQuery* pode ser observado a seguir.

Código 65: Exemplo de utilização do comando *ExecuteNonQuery*

```
1 SqlCommand oCmd = new SqlCommand("UPDATE Products SET UnitPrice=UnitPrice*1.1");
2 oCmd.Connection = oConn;
3 oCmd.CommandText = "UPDATE Products SET UnitPrice=UnitPrice*1.05";
4 oCmd.ExecuteNonQuery();
```

12.8.2 O método *ExecuteScalar*

O método *ExecuteScalar* é utilizado para comandos que retornam valores escalares, ou seja, valores únicos. Em geral é utilizado para comandos que retornam uma contagem de registros ou que executam ao comando de agregação no banco de dados.

Este comando pode retornar qualquer tipo de dado.

Um exemplo de utilização do comando *ExecuteScalar* pode ser observado a seguir.

Código 66: Exemplo de utilização do comando *ExecuteScalar*

```
1 SqlCommand oCmd = new SqlCommand("UPDATE Products SET UnitPrice=UnitPrice*1.1");
2 oCmd.Connection = oConn;
3 oCmd.CommandText = "SELECT COUNT(*) FROM Products";
4 int iNumProdutos;
5 iNumProdutos = oCmd.ExecuteScalar();
```

12.8.3 O método *ExecuteReader*

O método *ExecuteReader* é utilizado para executar *querys* que retornam um conjunto de dados. Este método tem como resultado um objeto do tipo *SqlDataReader*.

A classe *SqlDataReader* representa um cursor aberto no banco de dados com os dados retornados pela query informada no objeto de comando.

Para lermos os dados de um *DataReader*, é necessário executamos o método *Read*. Este método retorna verdadeiro caso um registro tenha sido lido do cursor do banco de dados, e falso quando não houver mais registros a serem lidos. É necessário chamar o método Read pelo menos uma vez, pois o cursor aberto não fica posicionado no primeiro registro.

Como o *DataReader* mantém um cursor aberto com o banco de dados, não é possível realizar nenhuma operação no banco de dados (utilizando a conexão utilizada pelo *DataReader*) enquanto o *DataReader* estiver aberto. Por tanto, é necessário fechar o *DataReader* imediatamente após a sua utilização.

Um exemplo de utilização do método *ExecuteReader* e da classe *DataReader* pode ser observado a seguir.

Código 67: Exemplo de utilização do comando *ExecuteReader*

```
1 SqlCommand oCmd = new SqlCommand("SELECT ProductName, ProductId FROM Products",
      oConn);
2 SqlDataReader oDr;
3 oDr = oCmd.ExecuteReader();
4 while (oDr.Read()) {
5         Debug.WriteLine(oDr("ProductName").ToString());
6 }
```

12.9 Passando parâmetros

É possível passar parâmetros para os objetos da classe *SqlCommand*. Para indicarmos parâmetros nas *querys* informadas neste objeto, utilizamos o símbolo @ como prefixo para indicar um parâmetro. Esta sintaxe pode variar de acordo com o banco de dados utilizado (o Oracle utiliza ":"por exemplo).

Depois de indicar os parâmetros na query, é preciso adicionar objetos do tipo *SqlParameter* na coleção de parâmetros do *SqlCommand*. A coleção de parâmetros pode ser acessada através da propriedade Parameters do objeto de comando.

Um exemplo de criação de parâmetros pode ser observado a seguir.

Código 68: Exemplo de utilização de parâmetros

```
1  SqlConnection oConn = new SqlConnection(sConnString);
2  SqlCommand oCmd = new SqlCommand();
3  oCmd.Connection = oConn;
4  oCmd.CommandText = "UPDATE Products " + " SET UnitPrice=@UnitPrice " + " WHERE
       ProductId=@ProductId";
5  SqlParameter oParam = new SqlParameter("@UnitPrice", 1.5);
6  oCmd.Parameters.Add(oParam);
7  oParam = new SqlParameter();
8  oParam.ParameterName = "@ProductId";
9  oParam.DbType = DbType.Int32;
10 oParam.Value = 1;
11 oCmd.Parameters.Add(oParam);
12 oCmd.ExecuteNonQuery();
```

12.10 O que é um *DataSet*?

O *DataSet* é uma classe capaz de armazenar múltiplos resultados tabulares em uma mesma estrutura. O *DataSet* é composto por estruturas chamadas *DataTables* que representam estes resultados tabulares.

Para extrairmos dados da base de dados e preenchermos o *DataSet* utilizamos a classe *DataAdapter*. Esta classe é capaz de executar os quatro comandos básicos de um banco de dados (*Insert, Update, Delete, Select*) sendo capaz de executar estas operações sobre os dados do *DataSet*.

12.11 O que é um *DataAdapter* ?

O *DataAdapter* é a classe responsável por fazer a interação entre a base de dados e o *DataSet*. Ela possui quatro propriedades que representam os quatro comandos principais que utilizamos para interagir com o banco de dados.

Para realizar a extração de dados do banco de dados para o *DataSet*, o *DataAdapter* usa o comando de *select*, contido na propriedade *SelectCommand*.

Após extrairmos os dados para o *DataSet*, podemos modificar estes dados (que estão armazenados em memória). À medida que modificamos os dados do *DataSet*, este faz uma marcação

nas alterações que fazemos, marcando as linhas inseridas como inseridas, modificadas como modificados e excluídos como excluídos. Quando concluímos as alterações, é possível chamar o *DataAdapter* novamente para que ele execute para cada linha modificada o comando correspondente a modificação realizada.

12.12 Criando um *DataSet* e um *DataAdapter*

Quando criamos um *DataAdapter* é possível informar uma *query* e uma conexão para a extração dos dados. O *SqlCommand* referente à propriedade *SelectCommand* é criado automaticamente. Os outros comandos devem ser criados manualmente.

Um exemplo de criação de um objeto *DataAdapter* pode ser observado a seguir.

Código 69: Criando um *DataSet* e um *DataAdapter*

```
1 SqlDataAdapter daProduct = new SqlDataAdapter("SELECT * FROM Products", oConn);
2 SqlDataAdapter daOrders = new SqlDataAdapter();
3 SqlCommand oCmd = new SqlCommand("SELECT * FROM Orders", oConn);
4 daOrders.SelectCommand = new SqlCommand(oCmd);
```

12.13 Criando e preenchendo um *DataSet*

Para criar um novo *DataSet* basta utilizar o comando New e criar um novo objeto. Para preencher um *dataset* utilizando um *DataAdapter*, devemos utilizar o método *Fill* do *DataAdapter*, informando o *DataSet* e nome da tabela a ser criada no *DataSet*.

Um exemplo de criação de um *DataSet* e utilização de um *DataAdapter* podem ser observados a seguir.

Código 70: Criando e preenchendo um *DataSet*

```
1 SqlDataAdapter daProduct = new SqlDataAdapter("SELECT * FROM Products", oConn);
2 DataSet ds = new DataSet();
3 daProduct.Fill(ds, "Products");
```