CURSO DE C#

por

Carlos Vamberto

Módulo I

Introdução ao .NET com C#

Dedicatória

À minha esposa Iara Bezerra e a minha filha Karlla Oksana

Agradecimentos

Ao grupo de estudo .NUG (.NET User Group - http://www.vsoft.com.br/dotnug), pelo incentivo de realizar um trabalho ao público voltado a tecnologia .NET.

Ao meu pai, Carlos Vamberto, pelo apoio didático e lógico deste material.

Prefácio

Imaginemos um mundo que em poucos intervalos de tempo esteja sempre evoluindo de forma veloz, onde os poucos que conseguem conviver naturalmente com essa evolução são considerados "gurus".

Agora vamos tornar este mundo imaginário em um mondo real, concreto. Este mundo a que nos refirimos é a TI – Tecnologia da Informação, composta de várias divisões e entre elas a do desenvolvimento de soluções de sistemas.

Julgamos que este material irá ajudar os interessados no conhecimento do topo da tecnologia de desenvolvimento de soluções, vez que ele fornecerá uma visão básica do .NET e conhecimentos necessários da principal linguagem de programação para esta tecnologia que é o C# (C Sharp da Microsoft).

Faremos uma abordagem da estrutura da linguagem em POO (Programação Orientada a Objeto) usando o C#, ademais das particularidades das Classes do Framework .NET e seu uso geral.

Sumário

Dedicatória **2**
Agradecimentos **2**
Prefácio **2**
Sumário **3**
Capítulo 1 – Introdução **4**
O que é .NET **4**
Conectando seu mundo **4**
.NET FrameWork **4**
Common Language Runtime (CLR) **5**
Compilação **6**
Captítulo 2 – Introdução ao C# **6**
Construindo a primeira aplicação em C# (Alô Mundo) **7**
Construindo uma aplicação com parâmetros **7**
Construindo uma aplicação com entrada interativa **8**
Tipos de Dados **8**
Boolean – Tipo Lógico **8**
Vetores **11**
Capítulo 3 - Trabalhando com Comandos Condicionais **13**
If … else **13**
switch…case **14**
Capítulo 4 - Trabalhando com Comandos de Repetição **15**
while **15**
do … while **16**
for **17**
for each **17**
Capítulo 5 - Métodos **18**
Capítulo 6 - Namespace **20**
Capítulo 7 - Classes **22**
Introdução **22**
Herança **23**
Capítulo 8 - Polimorfismo **25**
Capítulo 9 - Struts (Suportes) **26**
Capítulo 10 - Interface **27**
Capítulo 11 – Tratamento de Exceções **28**
Sintaxe de Tratamento do uso do try…catch **28**
Capítulo 12 - Introdução a ADO.NET (Banco de Dados) **29**
Vantagens do ADO.NET **29**
Classes do ADO.NET **30**
Esquema **30**
Provedores de Acesso a Dados **30**
OleDb **30**
SqlClient **31**
DataSet **31**
Conexão com Banco de Dados **32**
OleDB **32**
SqlClient **32**
Trabalhando com SqlCommand e SqlDataReader **32**
Incluindo Dados **34**
Alterando Dados **35**
Excluindo Dados **37**

Capítulo 1 – Introdução

O que é .NET

.NET é uma plataforma de software que conecta informações, sistemas, pessoas e dispositivos. A plataforma .NET conecta uma grande variedade de tecnologias de uso pessoal,de negócios, de telefonia celular a servidores corporativos, permitindo assim, o acesso rápido a informações importantes onde elas forem necessárias e imprescindíveis.

Desenvolvido sobre os padrões de Web Services XML, o .NET possibilita que sistemas e aplicativos, novos ou já existentes, conectem seus dados e transações independente do sistema operacional(SO) instalado, do tipo de computador ou dispositivo móvel que seja utilizado e da linguagem de programação que tenha sido utilizada na sua criação.

O .NET é um "ingrediente" sempre presente em toda a linha de produtos da Microsoft, oferecendo a capacidade de desenvolver, implementar, gerenciar e usar soluções conectadas através de Web Services XML, de maneira rápida, barata e segura. Essas soluções permitem uma integração mais ágil entre os negócios e o acesso rápido a informações a qualquer hora, em qualquer lugar e em qualquer dispositivo.

Conectando seu mundo

A idéia primordial por trás do Microsoft .NET é uma mudança de foco na informática: passa de um mundo de aplicativos Web sites e dispositivos isolados para uma infinidade de computadores, dispositivos, transações e serviços que se conectam diretamente e trabalham em conjunto para fornecerem soluções mais amplas e ricas.

As pessoas terão o controle sobre como, quando e que informações serão fornecidas a elas. Os computadores, sistemas e serviços serão capazes de colaborar e operar entre si, simultaneamente, em favor do usuário, enquanto as empresas poderão oferecer seus produtos e serviços aos clientes certos, na hora certa, da forma certa, combinando processos de maneira muito mais granular do que seria possível até hoje.

.NET FrameWork

Podemos dizer que o .NET Framework consiste em 3 partes:

- **CLR (Common Language Runtime) – Responsável pela interface entre o código e o sistema operacional;**
- **Classes do Framework - Todas as linguagens que usam a tecnologia .NET usam as mesmas classes;**
- **ASP.NET - Fornece o acesso direto a toda a linguagem VB.NET(Visual Basic) e/ou C# a partir de uma plataforma de "scriptação".**

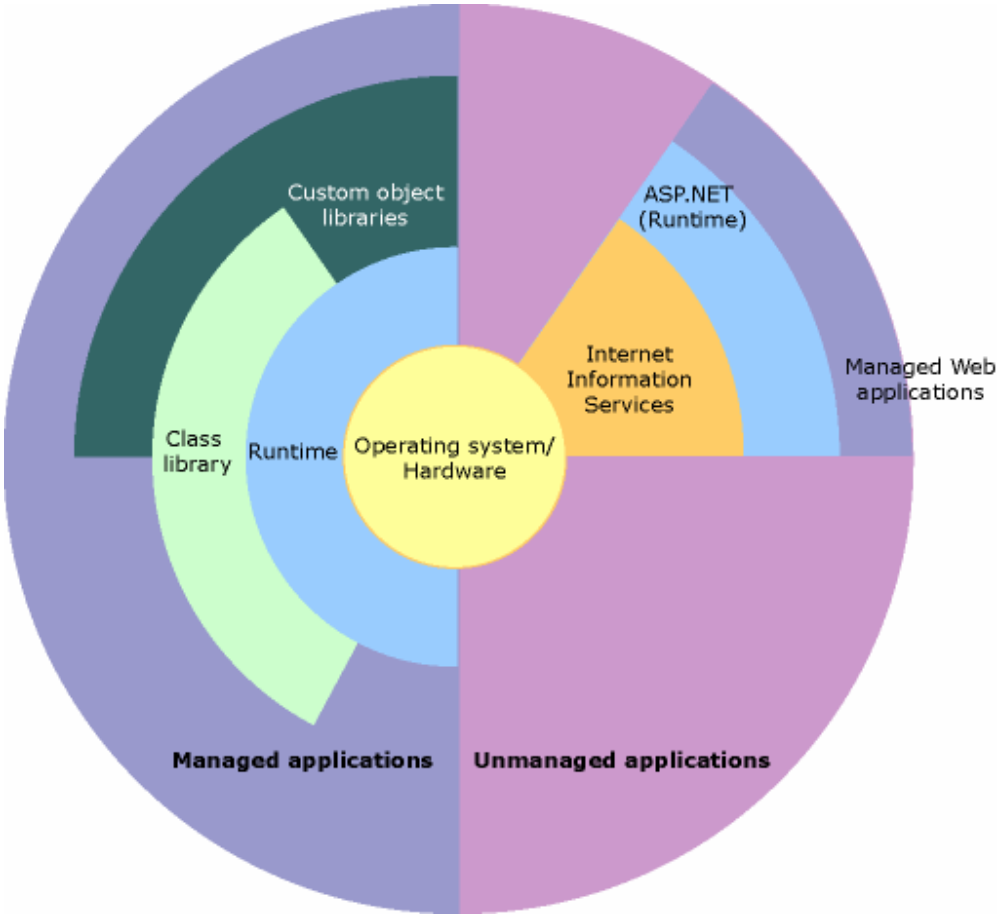
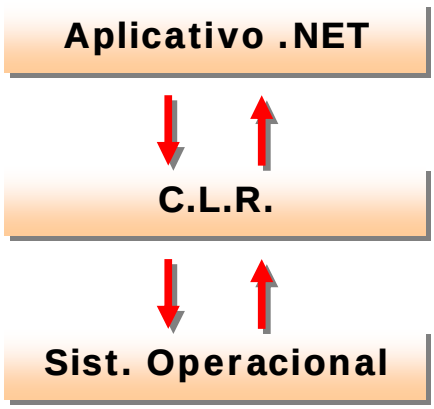


Ilustração 1 - Contexto do .NET Framework

Common Language Runtime (CLR)

Nessa nova tecnologia da Microsoft, o compilador não gera código nativo, ou seja, ele gera um código intermediário para qualquer SO. O código passa a ser compilado para ser rodado dentro do Interpretador CLR. Com isto, o desenvolvedor poderá escolher uma entre as várias linguagens de programação que trabalham com essa tecnologia.



Compilação

Qualquer linguagem que usa a tecnologia .NET, realiza o mesmo esquema de compilação, ou seja, ao compilar um código, gera-se um arquivo compilado para uma linguagem intermediária – MSIL (MicroSoft Intermediate Language). Esse arquivo gerado é chamado de Assenbly, podendo ter duas extensões: EXE ou DLL.

Quando o arquivo é executado, o JIT (Just-In-Time) converte este programa em código de máquina para ser rodado sobre o SO em que o CLR está rodando.

Desta forma, o MSIL gera o mesmo arquivo binário para qualquer plataforma que tiver o CLR, que por sua vez converte esse arquivo para código de máquina compatível ao SO corrente.

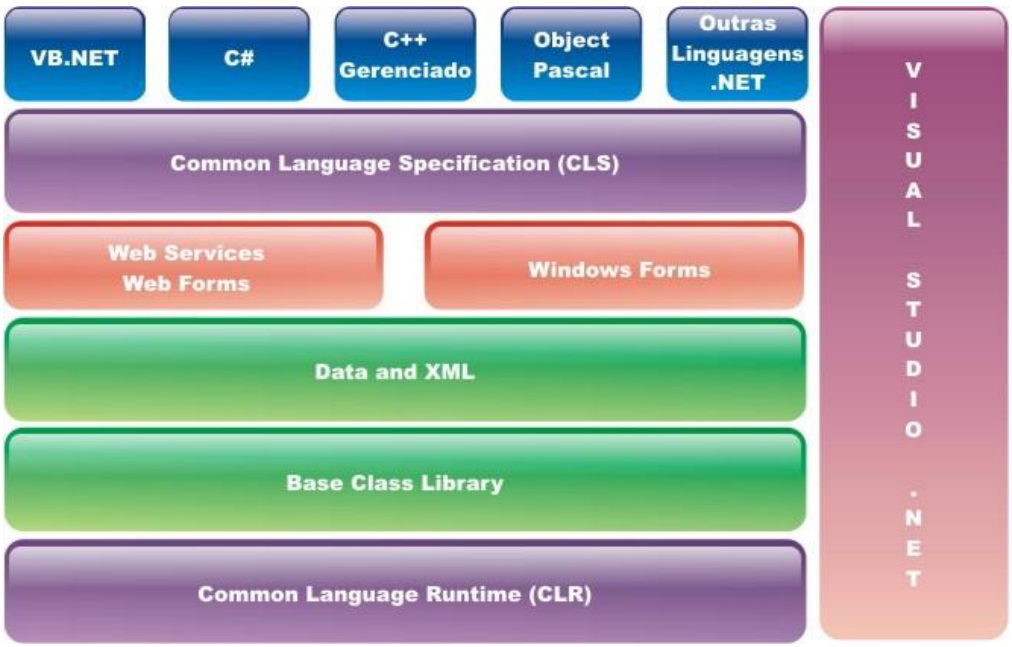


Ilustração 2 - Arquitetura do .NET

Daí se dá a idéia de Portabilidade, que consiste em procedida a compilação, o arquivo pode ser executado em várias plataformas de SO (Ex: Windows, Linux, MacOS, etc).

Captítulo 2 – Introdução ao C#

A Microsoft define o C# como a principal linguagem de programação para uso da tecnologia .NET. Por ser uma derivação da linguagem C++, sem as suas limitações, e é uma linguagem bastante simples de se implementar.

NOTA: O "Web Matrix" é uma ferramenta freeware para construção de sites ASP.NET feito totalmente em C#.

Construindo a primeira aplicação em C# (Alô Mundo)

Iremos construir agora uma simples aplicação com o C# do famoso "Alô Mundo !!!" que encontramos na maioria das linguagens.

Não se preocupe com a implementação, o objetivo é mostrar a eficiência do C#, vez que em tópicos posteriores você se familiarizará com essa linguagem. Ok!

1. **Crie uma pasta na unidade C: com o nome C:\CSHARP e dentro desta pasta crie uma subpasta com o nome C:\CSHARP\ALOMUNDO ;**
2. **Abra o prompt do DOS;**
3. **Entre na pasta criada com o comando: CD \CSHARP\ALOMUNDO + <ENTER>;**
4. **Digite EDIT alomundo.cs+ <ENTER>;**
5. **Digite a seguinte estrutura de código:**

```
using System;

class alomundo
{
   public static void Main() // Veja que no interior do parênteses não contém argumento.
   {
      Console.WriteLine("Alo mundo!!!");
   }
}
```

Local do Arquivo no CD: CD-ROM:\curso\A Alomundo\alomundo.cs

6. **Saia do editor salvando o projeto;**
7. **Compile o código usando o compilador do C#, CSC.EXE, digitando o seguinte comando no prompt do DOS:**

```
C:\CSHARP\ALUMUNDO> csc alomundo.cs+ <ENTER>
```

8. **Digite o comando DIR+ <ENTER> para ver se foi criado o arquivo ALOMUNDO.EXE**
9. **Para executar a aplicação digite "alomundo"+ <ENTER>**

NOTA: Caso o comando CSC não responda, é que ele não está no PATH (caminho de busca) do Windows. Para adicionar use o comando:

```
PATH %path%;C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
```

Suponha que o CSC.EXE se encontre nesta pasta.

Veja que esta aplicação criada só encontramos saída de dados com o camando WriteLine().

Construindo uma aplicação com parâmetros

Agora iremos criar uma pequena aplicação que terá parâmetros(argumentos) de entrada e exibirá o conteúdo deste argumento.

```
using System;

class Sistema
{
   public static void Main(String[] args)
   {
      Console.WriteLine("Seja bem vindo, {0}!", args[0]);
      Console.WriteLine("Este é o curso de C# por Carlos Vamberto");
   }
}
```

Local do Arquivo no CD: CD-ROM:\curso\B Parametros\argumentos.cs

Construindo uma aplicação com entrada interativa

Com esta aplicação saberemos como criar uma aplicação do tipo console(no teclado – estilo DOS) com entrada de dados interativos (você participa).

```
using System;

class entrada
{
   public static void Main()
   {
      Console.Write("Digite seu nome: ");
      Console.Write("Bem vindo, {0}",Console.ReadLine());
   }
}
```

Local do Arquivo no CD Anexo: CD-ROM:\curso\C Entrada Interativa\entrada.cs

Tipos de Dados

Nesta seção passaremos a conhecer os tipos de dados encontrados nas linguagens de programação do .NET

Boolean – Tipo Lógico

Este tipo aceita apenas dois valores: `false` (falso) ou `true` (verdadeiro)

Veja exemplo da declaração da variável e a atribuição de valores:

```
using System;

class logicos
{
      public static void Main()
      {
            bool ativo = true;
            bool inativo = false;

            Console.WriteLine("Aqui mostrará o valor ativo  : {0}",ativo);
            Console.WriteLine("Aqui mostrará o valor inativo: {0}",inativo);
      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\D Tipos de Dados\a_boolean.cs

Veja esta tabela contendo alguns tipos inteiros, tamanho e seus intervalos.

Tipo	bits	Intervalo
sbyte	8	-128 até 127
byte	8	0 até 255
short	16	-32768 até 32767
ushort	16	0 até 65535
int	32	-2147483648 até 2147483647
uint	32	0 até 4294967295
long	64	-9223372036854775808 até 9223372036854775807
ulong	64	0 até 18446744073709551615
char	16	0 até 65535

Agora a tabela de tipos Reais (pontos flutuantes e tipos decimais)

Tipo	bits	Precisão	Intervalo
float	32	7 dígitos	1.5×10^{-45} até 1.5×10^{38}
double	64	15-16 dígitos	5.0×10^{-324} até 1.7×10^{308}
decimal	128	28-29 decimal	1.0×10^{-28} até 7.9×10^{28}

Operadores

Categoria	Operador(es)	Associação
Primário	(x) x.y f(x) a[x] x++ x— new typeof sizeof checked unchecked	left
Unário	+ - ! ~ ++x --x (T)x	left
Multiplicidade	* / %	left
Aditivo	+ -	left
Substituição	<< >>	left
Relacional	< > <= >= is	left
Igualdade	== !=	right
AND Condicional	&&	left
OR Condicional	||	left
XOR Condicional	^	left

Exemplo trabalhando com operadores unários

```
using System;

class unario
{
	public static void Main()
	{
		// Declaração das variáveis
		int unario = 0;
		int preIncremento;
		int preDecremento;
		int posIncremento;
		int posDecremento;
		int positivo;
		int negativo;
		sbyte bitNao;
		bool logNao;

		// Início do código
		preIncremento = ++unario;
```

```
            Console.WriteLine("Pré-incremento       : {0}",preIncremento);
            Console.WriteLine("Unário               : {0}",unario);
            Console.WriteLine(" ");

            preDecremento = --unario;
            Console.WriteLine("Pré-decremento       : {0}",preDecremento);
            Console.WriteLine("Unário               : {0}",unario);
            Console.WriteLine(" ");

            posDecremento = unario--;
            Console.WriteLine("Pós-decremento       : {0}",posDecremento);
            Console.WriteLine("Unário               : {0}",unario);
            Console.WriteLine(" ");

            posIncremento = unario++;
            Console.WriteLine("Pós-incremento       : {0}",posIncremento);
            Console.WriteLine("Unário               : {0}",unario);
            Console.WriteLine(" ");

            Console.WriteLine("Valor Final do Unário: {0}",unario);
            Console.WriteLine(" ");

            positivo = -posIncremento;
            Console.WriteLine("Positivo             : {0}",positivo);

            negativo = +posIncremento;
            Console.WriteLine("Negativo             : {0}",negativo);
            Console.WriteLine(" ");

            bitNao = 0;
            bitNao = (sbyte)(~bitNao);
            Console.WriteLine("Bitwise              : {0}",bitNao);
            Console.WriteLine(" ");

            logNao = false;
            logNao = !logNao;
            Console.WriteLine("Não Lógico           : {0}",logNao);
            Console.WriteLine(" ");

      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\D Tipos de Dados\b_unarios.cs

Exemplo de trabalhando com operadores binários:

```
using System;
class Binary
{
      public static void Main()
      {
            int x, y, resultado;
            float floatResult;

            x = 7;
            y = 5;

            resultado = x+y;
            Console.WriteLine("x+y: {0}", resultado);

            resultado = x-y;
            Console.WriteLine("x-y: {0}", resultado);

            resultado = x*y;
            Console.WriteLine("x*y: {0}", resultado);
```

```
            resultado = x/y;
            Console.WriteLine("x/y: {0}", resultado);

            floatResult = (float)x/(float)y;
            Console.WriteLine("x/y: {0}", floatResult);

            resultado = x%y;
            Console.WriteLine("x%y: {0}", resultado);

            resultado += x;
            Console.WriteLine("resultado+=x: {0}", resultado);
      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\D Tipos de Dados\c_binarios.cs

Vetores

Trabalhar com vetores é muito similar na maioria das linguagens de programação. No C# é simples declarar e atribuir valores a vetores.

A sintaxe para a declaração de um *array* é bem simples, coloca-se o tipo desejado e em seguida os colchetes abrindo e fechando, o nome da variável e a alocação de tamanho do vetor. O exemplo abaixo declara um *array* de string unidimensional com 3 posições.

```
using System;

class teste
{
      public static void Main()
      {
            // Declaração
            string[] vetNome = new string[3];

            // Atribuição
            vetNome[0] = "Ana";
            vetNome[1] = "Pedro";
            vetNome[2] = "Maria";

            for (int x=0;x<=2;x++)
            {
                  Console.WriteLine("Nome {0} = {1}",x,vetNome[x]);
            }
      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\E Vetores\introducao1.cs

Também podemos construir *arrays* sem definir o tamanho inicial e atribuir depois os valores para cada elemento como no exemplo abaixo:

```
using System;

class teste
{
      public static void Main()
      {
            string[] vetNome = {"Ana","Pedro","Maria"};

            for (int x=0;x<=2;x++)
            {
                  Console.WriteLine("Nome {0} = {1}",x,vetNome[x]);
            }
```

```
        }
}
```

curso\E Vetores\introducao2.cs

No código abaixo você verá alguns exemplos trabalhando com vetores.

```
using System;
class Array
{
        public static void Main()
        {
                int[] myInts = { 5, 10, 15 };
                bool[][] myBools = new bool[2][];

                myBools[0] = new bool[2];
                myBools[1] = new bool[1];

                double[,] myDoubles = new double[2, 2];
                string[] myStrings = new string[3];

                Console.WriteLine("myInts[0]: {0}, myInts[1]: {1}, myInts[2]: {2}", myInts[0],
                        myInts[1], myInts[2]);

                myBools[0][0] = true;
                myBools[0][1] = false;
                myBools[1][0] = true;

                Console.WriteLine("myBools[0][0]: {0}, myBools[1][0]: {1}", myBools[0][0],
                        myBools[1][0]);

                myDoubles[0, 0] = 3.147;
                myDoubles[0, 1] = 7.157;
                myDoubles[1, 1] = 2.117;
                myDoubles[1, 0] = 56.00138917;

                Console.WriteLine("myDoubles[0, 0]: {0}, myDoubles[1, 0]: {1}", myDoubles[0,
                        0], myDoubles[1, 0]);

                myStrings[0] = "Joe";
                myStrings[1] = "Matt";
                myStrings[2] = "Robert";

                Console.WriteLine("myStrings[0]: {0}, myStrings[1]: {1}, myStrings[2]: {2}",
                        myStrings[0], myStrings[1], myStrings[2]);
        }
}
```

Local do Arquivo no CD: CD-ROM:\curso\E Vetores\vetores.cs

Capítulo 3 - Trabalhando com Comandos Condicionais

Neste capítulo iremos aprender a trabalhar com condicionais para tratar partes de nossos códigos.

If ... else

A condicional mais usada em linguagens de programação é o if(se) por ser fácil sua manipulação.

A sintaxe desta condicional é:

```
if (<lista de condições>)
{
	<comandos>;
}
Else // Senão
{
	<comandos>;
}
```

Nas listas de condições, estas devem ser intercaladas por AND (&&) ou OR (||)...

Veja no exemplo abaixo:

```
using System;
class IfSelect
{
	public static void Main()
	{
		string myInput;
		int myInt;
		Console.Write("Entre com um número e pressione ENTER: ");
		myInput = Console.ReadLine();
		myInt = Int32.Parse(myInput);

		// Decisão simples com brackets
		if (myInt > 0)
		{
			Console.WriteLine("Seu número {0} é maior que zero.", myInt);
		}
		// Decisão simples sem brackets
		if (myInt < 0)
			Console.WriteLine("Seu número {0} é menor que zero.", myInt);

		// Decisão com “Senão”
		if (myInt != 0)
		{
			Console.WriteLine("Seu número {0} não é igual a zero.", myInt);
		}
		else
		{
			Console.WriteLine("Seu número {0} é igual a zero.", myInt);
		}

		// Múltipla decisão com “E” e “OU”
		if (myInt < 0 || myInt == 0)
		{
			Console.WriteLine("Seu número {0} é menor ou igual a zero.", myInt);
		}
		else if (myInt > 0 && myInt <= 10)
```

```
            {
                  Console.WriteLine("Seu número {0} está entre 1 e 10.", myInt);
            }
            else if (myInt > 10 && myInt <= 20)
            {
                  Console.WriteLine("Seu número {0} está entre 11 e 20.", myInt);
            }
            else if (myInt > 20 && myInt <= 30)
            {
                  Console.WriteLine("Seu número {0} está entre 21 e 30.", myInt);
            }
            else
            {
                  Console.WriteLine("Seu número {0} é maior que 30.", myInt);
            }
      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\F Condicionais\a_se.cs

switch...case

Este é um tipo de condicional que verifica várias expressões para uma única comparação. Sua sintaxe é:

```
switch (<variável ou propriedade>) {
      case <valor1>:
            ...
            <expressões>
            ...
      case <valor2>:
            ...
            <expressões>
            ...
      default
            ...
            <expressões>
            ...
}
```

No exemplo abaixo faremos um programa que sempre ficará dentro de um laço através de *labels*(rótulos) como em linguagens estruturais como o VB, onde o C# também aceita.

```
using System;
class SwitchSelect
{
   public static void Main()
   {
      string myInput;
      int myInt;

      // Label BEGIN
      begin:
         Console.Write("Favor entrar com um número entre 1 e 3: ");
      myInput = Console.ReadLine();
      myInt = Int32.Parse(myInput);

      // switch com tipo inteiro
      switch (myInt)
      {
         case 1:
            Console.WriteLine("Seu número é {0}.", myInt);
            break;
         case 2:
```

```
            Console.WriteLine("Seu número é {0}.", myInt);
            break;
        case 3:
            Console.WriteLine("Seu número é {0}.", myInt);
            break;
        default:
            Console.WriteLine("Seu número {0} não está entre 1 e 3.", myInt);
            break;
    }

    // Label DECIDE
    decide:
        Console.Write("Digite \"continuar\" para continuar or \"sair\" para sair: ");
    myInput = Console.ReadLine();
    // switch com tipo string
    switch (myInput)
    {
        case "continuar":
            goto begin;
        case "sair":
            Console.WriteLine("Tchau.");
            break;
        default:
            Console.WriteLine("Sua entrada {0} está incorreta.", myInput);
            goto decide;
    }
  }
}
```

Local do Arquivo no CD: **CD-ROM:\curso\F Condicionais\b_caso.cs**

Capítulo 4 - Trabalhando com Comandos de Repetição

Existe vários tipos de comandos de repetição e eles são:

while

Sintaxe:

```
while (<condição>)
{
    ...
    <expressões>
    ...
}
```

No exemplo abaixo, faremos um loop usando o comando *while*, para escrever números entre 0 e 9.

```
using System;
class WhileLoop
{
    public static void Main()
    {
        int myInt = 0;
        while (myInt < 10)
        {
            Console.Write("{0} ", myInt);
            myInt++;
        }
        Console.WriteLine();
    }
```

```
}
```

Local do Arquivo no CD: **CD-ROM:\curso\G Loop\a_while.cs**

do ... while

O comando *DO..While* constrói um laço semelhante ao do *While*, mas ele só verifica a condição depois de executar as expressões.

Aqui está a sintaxe:

```
do
{
       ...
       <expressões>;
       ...
} while (<condicional>);
```

Agora veremos um exemplo usando simultaneamente comandos condicionais *switch* e *if* com o de repetição *do*.

```
using System;
class DoLoop
{
       public static void Main()
       {
              string myChoice;
              do
              {
                     // Imprime o Menu
                     Console.WriteLine("Meu Livro de Endereços\n");
                     Console.WriteLine("A - Adicionar");
                     Console.WriteLine("D - Deletar");
                     Console.WriteLine("M - Modificar");
                     Console.WriteLine("V - Visualizar");
                     Console.WriteLine("S - Sair\n");
                     Console.WriteLine("Choice (A,D,M,V,or S): ");

                     // Obtem a escolha do usuário
                     myChoice = Console.ReadLine();

                     // Trata a escolha do usuário
                     switch(myChoice)
                     {
                            case "A":
                            case "a":
                                   Console.WriteLine("Você escolheu ADICIONAR.");
                                   break;
                            case "D":
                            case "d":
                                   Console.WriteLine("Você escolheu DELETAR.");
                                   break;
                            case "M":
                            case "m":
                                   Console.WriteLine("Você escolheu MODIFICAR.");
                                   break;
                            case "V":
                            case "v":
                                   Console.WriteLine("Você escolheu Visualizar.");
                                   break;
                            case "S":
                            case "s":
```

```
                                Console.WriteLine("Tchau.");
                                break;
                        default:
                                Console.WriteLine("{0} não é uma opção válida", myChoice);
                                break;
                }

                // Dá uma pausa para permitir que o usuário veja o resultado
                Console.Write("Pressione ENTER para continuar...");
                Console.ReadLine();
                Console.WriteLine();
            } while (myChoice != "S" && myChoice != "s");
            // Repete até o usuário querer sair
  }
}
```

Local do Arquivo no CD: **CD-ROM:\curso\G Loop\b_do.cs**

for

Este comando de repetição, diferente do anterior que executa o procedimento várias vezes até que uma condição seja verdadeira, aqui será executado um número previamente determinado de vezes.

Veja a sintaxe:

```
for (<variável>, <condição>, <incremento>)
{
       ...
       <expressões>
       ...
}
```

Construa o exemplo abaixo:

```
using System;
class ForLoop
{
       public static void Main()
       {
              for (int i=0; i < 20; i++)
              {
                     if (i == 10)
                            break;
                     if (i % 2 == 0)
                            continue;
                     Console.Write("{0} ", i);
              }
              Console.WriteLine();
       }
}
```

Local do Arquivo no CD: **CD-ROM:\curso\G Loop\c_for.cs**

for each

Já neste comando de repetição podemos varrer todos os elementos de um vetor seja de objeto ou de tipos primitivos. Veja a sintaxe:

```
for each (<variável do tipo> in <vetor>)
{
       ...
       <expressões>
       ...
}
```

Veja agora um exemplo usando o comando *for...each* onde se varre um vetor de strings e escreve no console o conteúdo de cada posição:

```
using System;
class ForEachLoop
{
	public static void Main()
	{
		string[] names = {"Cheryl", "Joe", "Matt", "Robert"};
		foreach (string person in names)
		{
			Console.WriteLine("{0} ", person);
		}
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\G Loop\d_foreach.cs

Capítulo 5 - Métodos

Se você não sabe, o *Main()* que codificamos em nossos exemplos é um método de classe.

O método é na verdade a execução de uma rotina que está contida em uma classe. Desta forma, poderemos construir várias funcionalidades para uma única classe, para que esta fique mais fácil de futuras correções no nível de código.

Agora veja o código fonte da classe Aluno:

```
using System;

class Aluno
{
	string nome;

	public static void Main()
	{
		// Instanciando o objeto ALUNO
		Aluno aluno = new Aluno();

		// Executando o método setNome para colocar o nome do Aluno
		aluno.setNome("Carlos");

		// Escrevendo o nome do aluno no console usando o método getNome
		Console.WriteLine(aluno.getNome());
	}

	void setNome(string n)
	{
		nome = n;
	}

	string getNome()
	{
		return nome;
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\H Método\a_metodo.cs

Outro exemplo da classe *Aluno* sendo que esta será instanciada pela classe que conterá o método *Main()*.

```
using System;

class Aluno
{
	protected string nome;

	public void setNome(string n)
	{
		nome = n;
	}

	public string getNome()
	{
		return nome;
	}

	protected void delNome()
	{
		nome = "";
	}

	public void deleta()
	{
		this.delNome();
	}
}

class Metodo
{
	public static void Main()
	{
		// Instanciando o objeto ALUNO
		Aluno aluno = new Aluno();

		// Executando o método setNome para colocar o nome do Aluno
		aluno.setNome("Carlos");

		// Escrevendo o nome do aluno no console usando o método getNome
		Console.WriteLine(aluno.getNome());

		// Excluir o conteúdo do nome
		aluno.deleta();
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\H Método\b_metodo.cs

Perceba que os métodos da classe *Aluno* estão agora com a palavra reservada *public* no início. Isto faz com que o método seja "enxergado" por outro método.

Os métodos podem ser:

- **public – Métodos que permitem que outras classes possam acessá-los;**
- **private – Métodos que inibe que outras classes possam acessá-los;**
- **protected – Métodos que permitem que apenas a classe e as subclasses façam acesso a eles.**

NOTA: O *public, private* e *protected* podem ser usados também em atributos (propriedades) das classes como mostrado no exemplo acima no atributo *nome*.

Capítulo 6 - Namespace

Namespace **é um programa em C# que foi desenvolvido para ajudá-lo a organizar seus programas. Fornece também auxílio em evitar classes de mesmo nome dentro de um *Namespace*. É bom usar *Namesapace* em seu código, pois, ele facilitará a reutilização de um código já implementado em outro projeto.**

Um *namespace* não corresponde a um nome de arquivo ou diretório como no Java. Você pode renomear os arquivos e diretórios para ajudá-lo a organizar seu projeto mais sem a obrigatoriedade.

Vamos agora criar o nosso primeiro *namespace* usando o *alomundo* que criamos anteriormente:

```
using System;

// Criando o nosso primeiro name space com o nome "ns"
namespace ns
{
	// criando a classe "alomundo" dentro do namespace
	class alomundo
	{
		public static void Main()
		{
			Console.WriteLine("Alo mundo !!!");
		}
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\I Namespace\a_namespace.cs

Podemos também ter *namespace* dentro de outro *namespace* com as classes implementadas, conforme exemplo abaixo:

```
using System;

// Criando o nosso primeiro name space com o nome "ns"
namespace ns
{
	namespace brasil
	{
		// criando a classe "alomundo" dentro do namespace
		class alomundo
		{
			public static void Main()
			{
				Console.WriteLine("Alo mundo !!!");
			}
		}
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\I Namespace\b_namespace.cs

Também poderemos construir os *Namespaces* dentro de outro, sendo intercalados pelo ponto (.). Veja o exemplo que segue:

```
using System;

// Criando o nosso primeiro name space com o nome "ns"
```

```
namespace ns.brasil
{
	// criando a classe "alomundo" dentro do namespace
	class alomundo
	{
		public static void Main()
		{
			Console.WriteLine("Alo mundo !!!");
		}
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\I Namespace\c_namespace.cs

Agora vamos dar mais profissionalismo no uso do *Namespace*. Iremos construir um código que faz chamada a um *Namespace*.

```
using System;

// Criando o nosso primeiro name space com o nome "ns"
namespace ns
{
	namespace mensagem
	{
		// criando a classe "alomundo" dentro do namespace
		class alomundo
		{
			public static void alo()
			{
				Console.WriteLine("Alo mundo !!!");
			}
		}
	}

	class mandamensagem
	{
		public static void Main()
		{
			mensagem.alomundo.alo();
		}
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\I Namespace\d_namespace.cs

Agora construiremos um código que irá fazer uma chamada ao um *Namespace* através de diretiva:

```
// Diretivas
using System;
using ns.mensagem;

// Criando o nosso primeiro name space com o nome "ns"
namespace ns
{
	namespace mensagem
	{
		// criando a classe "alomundo" dentro do namespace
		class alomundo
		{
			public static void alo()
			{
				Console.WriteLine("Alo mundo !!!");
			}
		}
	}
```

```
}
class mandamensagem
{
	public static void Main()
	{
		alomundo.alo();
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\I Namespace\e_namespace.cs

Perceba que no exemplo acima, a classe *mandamensagem* fica fora do *Namespace*, e por isto é que faremos referencia a ele através do *using* no início do código.

Podemos também usar as diretivas com apelidos (alias) a uma determinada classe. Veja no exemplo abaixo:

```
using System;
using mens = ns.mensagem.alomundo; // Aqui está o apelido “mens”

// Criando o nosso primeiro name space com o nome "ns"
namespace ns
{
	namespace mensagem
	{
		// criando a classe "alomundo" dentro do namespace
		class alomundo
		{
			public static void alo()
			{
				Console.WriteLine("Alo mundo !!!");
			}
		}
	}
}
class mandamensagem
{
	public static void Main()
	{
		mens.alo();
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\I Namespace\f_namespace.cs

Capítulo 7 - Classes

Introdução

Desde que começamos este tutorial trabalhamos com classes. Agora você terá um sentido melhor na definição de uma classe e conhecer novos membros que compõe uma classe.

Cada classe necessita de um construtor (alocador de memória), e ela é automaticamente chamada quando a classe é instanciada criando um objeto. Os construtores não retornam valores e não impositivo ter o mesmo nome da classe.

Vamos criar agora uma classe *Aluno* com um construtor:

```
using System;

class Aluno
{
```

```
        string nome;

        // Criando o Método Construtor
        public Aluno(string n)
        {
                nome = n;
        }

        // Método setNome : Coloca valor no atributo nome
        public void setNome(string n)
        {
                nome = n;
        }

        // Método getNome : Retorna o valor do atributo nome
        public string getNome()
        {
                return nome;
        }
}

class exemplo
{
        public static void Main()
        {
                // Cria o objeto "alu" que é a instância da classe "Aluno"
                Aluno alu = new Aluno("Carlos");
                Console.WriteLine("Nome do aluno é {0}",alu.getNome());

                alu.setNome("Pedro");
                Console.WriteLine("Nome do aluno agora é {0}",alu.getNome());
        }
}
```

Local do Arquivo no CD: **CD-ROM:\curso\J Classes\a_classe.cs**

Herança

A herança é um dos conceitos primários de programação orientada a objeto. Isto faz com que o reuso de código seja feito em nível de classe.

Uma vez implementando uma classe herdada de outra, esta nova classe herdará todos os atributos e métodos da classe pai e, poderá, ainda, ter, também, seus próprios atributos e métodos.

```
using System;

// Criando a classe "Pessoa"
public class Pessoa
{
   public string nome;
   public int idade;

   // Criando o construtor "Pessoa"
   public Pessoa()
   {
       Console.WriteLine("Foi criando um Objeto Pessoa");
   }
}


// Criando a classe "Aluno" que herda "Pessoa"
public class Aluno : Pessoa
{
   public string matricula;
```

```
    // Criando o construtor "Aluno"
    public Aluno()
    {
        Console.WriteLine("Foi criado um Objeto Aluno");
    }
}

// Iniciando a aplicação
class exemplo
{
    public static void Main()
    {
     // Criando uma instância da classe "Pessoa"
        Pessoa pessoa = new Pessoa();
        pessoa.nome = "Carlos";
        pessoa.idade = 30;

        Console.WriteLine("Pessoa: Nome={0}; Idade={1}",pessoa.nome,pessoa.idade);
        Console.WriteLine();

        // Criando uma instãncia da classe "Aluno"
        Aluno aluno = new Aluno();
        aluno.nome = "Marcos";
        aluno.idade = 31;
        aluno.matricula = "1234";

        Console.WriteLine("Aluno: Nome={0}; Idade={1}; Matrícula={2}",
              aluno.nome,aluno.idade,aluno.matricula);
        Console.WriteLine();
    }
}
```

Local do Arquivo no CD: **CD-ROM:\curso\J Classes\b_heranca.cs**

Neste novo exemplo trabalharemos em um código que mostra uma herança entre classes que possuem construtores distintos:

```
using System;

// Criando a classe pública "Pessoa"
public class Pessoa
{
    // Atributos
    public string nome;

    // Construtor
    public Pessoa()
    {
        // Nada a fazer
    }

    // Construtor
    public Pessoa(string n)
    {
        this.nome = n;
    }
}

// Criando a classe pública "Aluno" que herda características da classe "Pessoa"
public class Aluno : Pessoa
{
    // Adiciona outro atributo
    public string matricula;
```

```
    // Construtor com base na herança
    public Aluno(string m, string n) :  base(n)
    {
        this.matricula = m;
    }

}


// Classe para instanciar os objetos e mostrar os resultados
class Exemplo
{
    public static void Main()
    {
        // Instanciando a classe Pessoa
        Pessoa pessoa = new Pessoa("Carlos");
        Console.WriteLine("Pessoa com nome {0}", pessoa.nome);

        // Instanciando a classe Aluno
        Aluno aluno = new Aluno("1234","Ana");
        Console.WriteLine("Aluno com nome {0} e matrícula {1}", aluno.nome, aluno.matricula);
    }
}
```

Local do Arquivo no CD: **CD-ROM:\curso\J Classes\C_heranca.cs**

Capítulo 8 - Polimorfismo

Um outro conceito preliminar na POO (Programação Orientada a Objeto) é o polimorfismo. Sua principal função é tornar possível criar métodos de mesmo nome em classes diferentes.

Veja um exemplo de polimorfismo usando herança de classes:

```
using System;

// Classe : Ponto
public class Ponto
{
	public virtual void desenhar()
	{
		Console.WriteLine("Método 'desenhar' do objeto 'Ponto'");
	}
}


// Classe : Linha
class Linha : Ponto
{
	public override void desenhar()
	{
		Console.WriteLine("Método 'desenhar' do objeto 'Linha'");
	}
}


// Classe : Circulo
class Circulo : Ponto
{
	public override void desenhar()
	{
		Console.WriteLine("Método 'desenhar' do objeto 'Circulo'");
	}
}
```

```
class teste
{
	public static void Main()
	{
		// Intância da classe Ponto
		Ponto ponto = new Ponto();
		ponto.desenhar();

		// Intância da classe Linha
		Linha linha = new Linha();
		linha.desenhar();

		// Intância da classe Circulo
		Circulo circulo = new Circulo();
		circulo.desenhar();
	}
}
```

Local do Arquivo no CD: CD-ROM:\curso\K Polimorfismo\a_polimorfismo.cs

Capítulo 9 - Struts (Suportes)

Um *strut* permite que você crie propriedades do tipo similar aos tipos internos da classe.

Veja um exemplo abaixo:

```
using System;

class Ponto
{
	public int x;
	public int y;

	// Construtor
	public Ponto(int x, int y)
	{
		this.x = x;
		this.y = y;
	}

	// Método Adicionar
	public Ponto Adicionar(Ponto ponto)
	{
		Ponto novoponto = new Ponto(0,0);

		novoponto.x = ponto.x + x;
		novoponto.y = ponto.y + y;

		return novoponto;
	}
}
class Exemplo
{
	public static void Main()
	{
		Ponto p1 = new Ponto(1,1);
		Ponto p2 = new Ponto(2,2);
		Ponto p3;

		p3 = p1.Adicionar(p2);
```

```
            Console.WriteLine("p3 = {0},{1}", p3.x, p3.y);
      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\L Struts\a_strut.cs

Perceba que no código acima no método *Adicionar* foi passado como parâmetro para outro objeto *Ponto.*

Capítulo 10 - Interface

Devemos entender uma *Interface* como uma classe, contudo sem qualquer implementação. A única coisa que contém na interface são as definições dos eventos, indexadores, métodos e/ou propriedades. A razão de ela existir é pelo fato dela poder ser herdada por classes e structs.

Então, por que as interfaces são boas se elas não tem funcionalidade na implementação? É porque elas são boas para unir arquiteturas plug-n-play (está presente, está funcionando) que usam componentes intercambiados. Com todos os implementos dos componentes intercambiados de alguma interface, eles pode ser usados sem a necessidade de uma programação extra. A interface força que cada componente exponha seus membros públicos que poderão ser usados.

Veja um breve exemplo abaixo que usa interface dentro de outra interface:

```
using System;

// Criando Interface : IInterfaceMae
interface IInterfaceMae
{
      void MetodoMae();
}

// Criando Interface : IMinhaInterface
interface IMinhaInterface : IInterfaceMae
{
      void MeuMetodo();
}

//Criando Classe para usar as interfaces através de herança
class Componente : IMinhaInterface
{
      public void MeuMetodo()
      {
            Console.WriteLine("Método do IMinhaInterface foi chamado");
      }

      public void MetodoMae()
      {
            Console.WriteLine("Método do IInterfaceMae foi chamado");
      }
}

// Criando Classe para testar a instância da classe que usa interface e o método.
class Testador
{
      public static void Main()
      {
            Componente componente = new Componente();
            componente.MeuMetodo();
            componente.MetodoMae();
      }
}
```

Local do Arquivo no CD: CD-ROM:\curso\M Interfaces\a_interface.cs

Capítulo 11 – Tratamento de Exceções

As exceções são tratamentos de erros que acontecem em seus programas para que não seja passado para o usuário final. Na maioria das vezes, você pode detectar quais locais no seu código que pode causar erros. Você tem sempre que está pensando em tratamentos de exceções, pois deixa seu programa mais seguro e sustentável.

Quando uma exceção deve ser implementada? Muitas vezes você é capaz de visualizar trechos de códigos que possam gerar algum erro como, por exemplo: Erro de I/O (E/S); erro de acesso a banco de dados; erros de acesso à memória; etc. Nestes trechos de códigos devemos colocar o Tratamento de Exceções para que seu programa não seja abortado e/ou que não passe uma informação técnica fora do alcance do usuário final.

Quando uma exceção ocorre, elas são chamadas de *thrown*. O *thrown* atual é um objeto que é derivado da classe *System.Exception*.

A classe *System.Exception* contém alguns métodos e propriedades para obter informações sobre o erro que foi gerado. Por exemplo o *Message* é uma propriedade que contém um resumo do erro gerado; *StackTrace* retorna informações do erro ocorrido; o método *ToString()* retorna uma descrição completa do erro ocorrido.

Existem vários tipos de Exceções. Por exemplo, na classe *System.IO.File* o método *OpenRead()*, poderá ter as seguintes exceções:

- ***SecurityException;***
- ***ArgumentException;***
- ***ArgumentNullxEception;***
- ***PathTooLongException;***
- ***DirectoryNotFoundException;***
- ***UnauthrizedAccessException;***
- ***FileNotFoundException;***
- ***NotSupportedException.***

Sintaxe de Tratamento do uso do try...catch

Quando fazemos um tratamento de exceção, podemos tratar vários erros em apenas uma estrutura. Os tratamentos mais ambíguos devem ficar entre os últimos e os mais específicos devem ficar entre os primeiros. Vejamos um exemplo usando o *System.IO.File*:

```
using System;
using System.IO;

class exemplo
{
   public static void Main()
   {
      try
      {
         // Isto causará uma exceção
         File.OpenRead("\\teste\\ArquivoNãoExistente.txt");
      }
      catch (FileNotFoundException e)
      {
```

```
            Console.WriteLine();
            Console.WriteLine("Erro pelo FileNotFoundException");
            Console.WriteLine("----------------------------------------");
            Console.WriteLine(e.ToString());
            Console.WriteLine();
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine();
            Console.WriteLine("Erro pelo DirectoryNotFoundException");
            Console.WriteLine("----------------------------------------");
            Console.WriteLine(e.ToString());
            Console.WriteLine();
        }
        catch (Exception e)
        {
            Console.WriteLine();
            Console.WriteLine("Erro pelo Exception");
            Console.WriteLine("----------------------------------------");
            Console.WriteLine(e.ToString());
            Console.WriteLine();
        }
    }
}
```

Local do Arquivo no CD: CD-ROM:\curso\N Exceptions\a_exception.cs

Capítulo 12 - Introdução a ADO.NET (Banco de Dados)

O ADO.NET é uma nova filosofia de conectividade com banco de dados. Ela não é compatível com a do ADO (Access Data Object). O ADO.NET é a última implementação da estratégia do *Universal Data Access* da Microsoft.

A Microsoft está posicionando o ADO.NET para ser a tecnologia preliminar do acesso dos dados para a estrutura do .NET.

A Microsoft tem dedicado um bocado de trabalho no acesso a dados. Ela tem realizado um grande número de mudanças na tecnologia de acesso a dados, e o ADO .NET não é outra coisa senão mais um destes avanços. Os desenvolvedores descobrirão toda uma nova forma de criar aplicações orientadas para banco de dados. Combinando o poder acesso a dados, XML e da .NET Framework, a Microsoft apresentou sua mais poderosa tecnologia de acesso a dados até hoje.

Vantagens do ADO.NET

Escalabilidade - pelo fato de o DataSet ser baseado em acesso a dados desconectados, por ter uma arquitetura baseada no XML, o tempo de manipulação dos dados no banco de dados torna-se mínimo. Portanto, mesmo com um número simultâneo de acesso maior, a aplicação consegue garantir uma boa escalabilidade;

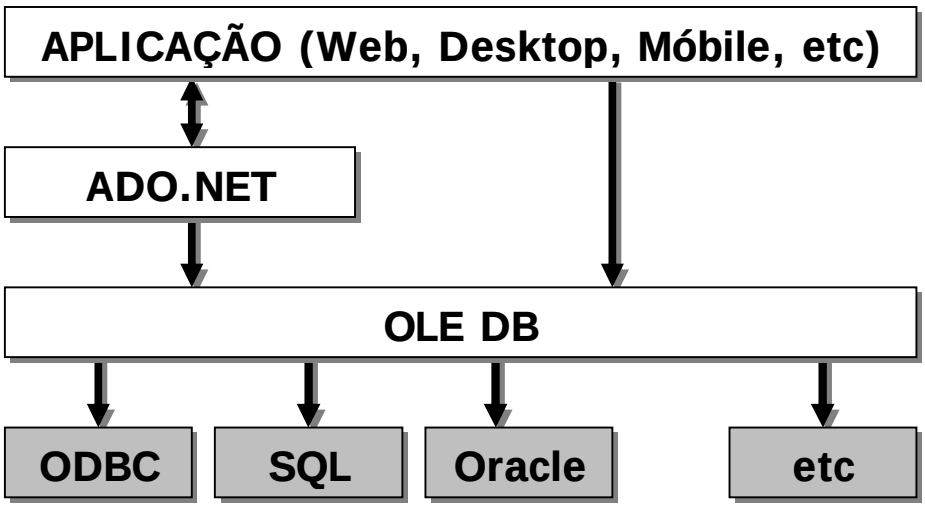
Performance - no ADO.NET a transmissão de dados é feita em XML, com isso é possível a comunicação com diversas plataformas e aplicativos;

Segurança - um firewall(bloqueadores de acessos externos) não consegue bloquear um arquivo texto. Portanto como o ADO.NET é baseado em XML, as portas dos firewalls não são mais problemas.

Classes do ADO.NET

- **System.Data**
- **System.Data.SqlClient**
- **System.Data.OleDb**
- **System.Data.SqlTypes**

Esquema



Provedores de Acesso a Dados

OleDb

A Plataforma .NET criou esta classe com vários provedores para o acesso de qualquer plataforma, como: SQL Server, Oracle e Access.

Este provedor pertence à classe System.Data.OleDb e suas principais Classes são:

- **OleDbConnection – Define a abertura da conexão;**
- **OleDbCommand – Define a instrução SQL a ser executada;**
- **OleDbDataReader – Define somente para leitura um conjunto de dados;**
- **OleDbDataAdapter – Define a conexão a ser usada para preencher um *DataSet* e representa um conjunto de comandos de dados.**

Esta classe possue os seguintes provedores:

- **SQLOLEDB – Microsoft OLE DB Provider for SQL Server;**
- **MSDAORA – Microsoft OLE DB Provider for Oracle;**
- **Microsoft.Jet.OLEDB.4.0 – OLE DB Provider for Microsft Jet.**

SqlClient

Este provedor é para o acesso exclusivo do SQL Server. Pertence a classe System.Data.SqlClient e suas principais Classes são:

- **SqlConnection – Define a abertura da conexão;**
- **SqlCommand – Define a instrução SQL a ser executada;**
- **SqlDataReader – Define somente para leitura um conjunto de dados;**
- **SqlDataAdapter – Define a conexão a ser usada para preencher um DataSet, e representa um conjunto de comandos de dados.**

NOTA: Se a aplicação desenvolvida utiliza o banco de dados SQLServer 7.0 ou superior, é altamente recomendável o uso do provedor SQL.

DataSet

O ADO.NET baseia-se no DataSet. Esse objeto é um conceito completamente novo para substituir o tradicional Recordset do ADO.

O DataSet é um armazenamento de dados simples residente na memória, que fornece um modelo de programação consistente de acesso a dados, independentemente do tipo de dados.

Diferentemente do RecordSet, o DataSet mantém conjuntos completos de dados, incluindo restrições, relacionamentos e até múltiplas tabelas de uma vez, todos estes armazenados na memória do computador.

Podemos afirmar também que os componentes do DataSet foram desenvolvidos para manipular dados com mais rapidez, sendo possível executar comandos de leitura, alteração de dados, *stored procedures* e enviar ou receber informações parametrizadas.

Já que todas as linhas da seleção são transmitidas de uma vez para a memória, é preciso ter cuidado neste aspecto importante para não comprometer a performance, sobrecarregando a memória e a rede.

Segundo o autor Américo Damasceno Junior o SELECT que você cria tem que ser muito bem estudado para termos um objeto tipo DataSet de, no máximo, uns 100 Kbytes (isso daria umas 1000 linhas de 100 dígitos mais ou menos). Mesmo assim, a uma velocidade de 3Kbytes/segundo (que é o máximo que geralmente se consegue numa hora boa de Internet com um modem de 56 Kbits/segundo), seriam necessários 30 segundos para o DataSet ir de uma maquina para outra.

O ideal é um objeto tipo DataSet de uns 5Kbytes (50 linhas de 100 dígitos mais ou menos) que para uma transmissão a 1Kbyte/segundo demandaria aceitáveis 5 segundos.

Conexão com Banco de Dados

Vamos agora ver alguns exemplos de conexão com Banco de dados usando o *OleDb* e o *SqlClient*.

OleDB

```
using System.Data.OleDb;

class banco
{
       public static void Main()
       {
              string strConexao = "Provider=SQLOLEDB;Server=Servidor;"
                     + "Database=NorthWind;User id=sa;pwd=Senha";

              OleDbConnection conn = new OleDbConnection(strConexao);

    conn.Open();
       }
}
```

SqlClient

```
using System.Data.SqlClient;

class banco
{
       public static void Main()
       {
              string strConexao = "DataSource=Servidor;User id=sa;pwd=Senha;"
                     + "Initial Catalog=Northwind";

              SqlConnection conn = new SqlConnection(strConexao);

    conn.Open();
       }
}
```

Trabalhando com SqlCommand e SqlDataReader

Vejamos um exemplo de conexão com uma base de dados em SQL Server onde armazenaremos os dados da consulta dentro de um *DataReader*. O banco de dados que escolhemos é o Northwind, pois ele é um banco de dados quem já vem com o SQL Server para demonstração.

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace MsdnAA
{
   class AdoNetApp
   {
       static void Main (string[] args)
       {
           string strConexao = "Server=wxp-01;Initial Catalog=Northwind;user id=sa";
```

```
            SqlConnection conn = new SqlConnection(strConexao);
            conn.Open();

            SqlCommand comm = new SqlCommand("select * from customers",conn);

            SqlDataReader dr = comm.ExecuteReader();

            for (int i = 0; i < dr.FieldCount; i++)
            {
                Console.Write("{0},", dr.GetName(i));
            }

            Console.WriteLine();

            while (dr.Read())
            {
                for (int i = 0; i <dr.FieldCount;i++)
                {
                      Console.Write("{0}, ",dr[i]);
                }
                Console.WriteLine();
            }
            dr.Close();
            conn.Close();
        }
    }
}
```

Local do Arquivo no CD: CD-ROM:\curso/O Database/a_database.cs

Iremos detalhar agora o mesmo exemplo anterior usando o OleDb.

```
using System;
using System.Data;
using System.Data.OleDb;

namespace MsdnAA
{
  class AdoNetApp
  {
    static void Main (string[] args)
    {
      string strConexao = "Provider=SQLOLEDB;Server=wxp-01;Database=Northwind;user id=sa";

      OleDbConnection conn = new OleDbConnection(strConexao);
      conn.Open();

      OleDbCommand comm = new OleDbCommand("select * from customers",conn);

      OleDbDataReader dr = comm.ExecuteReader();

      for (int i = 0; i < dr.FieldCount; i++)
      {
        Console.Write("{0},", dr.GetName(i));
      }

      Console.WriteLine();

      while (dr.Read())
      {
        for (int i = 0; i <dr.FieldCount;i++)
        {
          Console.Write("{0}, ",dr[i]);
        }
        Console.WriteLine();
      }
```

```
     dr.Close();
     conn.Close();
   }
  }
}
```

Local do Arquivo no CD: **CD-ROM:\curso/O Database/b_database.cs**

Incluindo Dados

Veremos um exemplo de inclusão de dados em uma tabela usando o objeto *SqlCommand* do SQL.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Incluir
{
      public static void Main()
      {
            // Criando a variáveis para inclusão da tabela CATEGORIAS
    string Nome;
            string Descricao;

            // Solicitando o Nome e a Descrição
            Console.WriteLine();
            Console.WriteLine("Inclusão de Categorias");
            Console.WriteLine("---------------------------------");
            Console.Write("Nome     : ");
            Nome = Console.ReadLine();
            Console.Write("Descrição: ");
            Descricao = Console.ReadLine();
            Console.WriteLine("---------------------------------");
            Console.WriteLine();
            Console.WriteLine("Fazento a inclusão");

            try
            {
                  // Criando a Conexão com o Servidor e o Banco de dados
                  string strCnx = "Server=wxp-01;Initial Catalog=Northwind;user id=sa";
                  string strCmm = "INSERT INTO Categories (CategoryName,Description)"
                        + " values ('" + Nome + "','" + Descricao + "')";

                  Console.WriteLine("String de Inclusão Criada:");
                  Console.WriteLine(strCmm);
                  Console.WriteLine();

                  Console.WriteLine("Fazendo a conexão com o banco e dados");
                  SqlConnection conn = new SqlConnection(strCnx);
                  conn.Open();

                  Console.WriteLine("Criando um comando de inclusão em SQL");
                  SqlCommand comm = new SqlCommand(strCmm,conn);

                  Console.WriteLine("Executando o comando de inclusão");
                  comm.ExecuteNonQuery();
                  Console.WriteLine();
                  Console.WriteLine("Incluído com sucesso");
                  Console.WriteLine();

                  conn.Close();
            }
            catch (Exception ex)
            {
                  Console.WriteLine();
```

```
                    Console.WriteLine("ERROR");
                    Console.WriteLine("-----------------");
                    Console.WriteLine("Erro ao tentar incluir");
                    Console.WriteLine(ex.ToString());
                    Console.WriteLine("-----------------");
             }
      }
}
```

curso/O Database/c_incluir.cs

Alterando Dados

Aqui faremos uma listagem de todas as categorias cadastradas da tabela *Categories* e escolheremos uma através do campo CategoryId para realizar sua alteração.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Incluir
{
      public static void Main()
      {
            // Criando a variáveis para alteração da tabela CATEGORIAS
            string Id;
            string Nome;
            string Descricao;

            string strCnx = "Server=wxp-01;Initial Catalog=Northwind;user id=sa";
            string strCmm;
            SqlConnection conn;
            SqlCommand comm;

            // Listando as categorias
            try
            {
                  strCmm = "select CategoryID,CategoryName,Description from Categories";
                  conn = new SqlConnection(strCnx);
                  conn.Open();

                  comm = new SqlCommand(strCmm, conn);

                  SqlDataReader dr = comm.ExecuteReader();
                  Console.WriteLine();
                  Console.WriteLine("L I S T A G E M");
                  Console.WriteLine("---------------");

                  while (dr.Read())
                  {
                        Console.Write("[{0}] [{1}] [{2}]",dr[0],dr[1],dr[2]);
                        Console.WriteLine();
                  }

                  Console.WriteLine("---------------");


                  conn.Close();

            }
            catch (Exception e)
            {
                  Console.WriteLine();
                  Console.WriteLine("ERROR");
                  Console.WriteLine("-----------------");
                  Console.WriteLine("Erro ao tentar listar");
```

```
                  Console.WriteLine(e.ToString());
                  Console.WriteLine("-----------------");
            }

            Console.Write("Digite o CategoryID para a alteração: ");
            Id = Console.ReadLine();

            // Solicitando o Nome e a Descrição
            Console.WriteLine();
            Console.WriteLine("Alteração de Categorias (Digite os novos valores)");
            Console.WriteLine("-----------------------------------------------------");
            Console.Write("Nome     : ");
            Nome = Console.ReadLine();
            Console.Write("Descrição: ");
            Descricao = Console.ReadLine();
            Console.WriteLine("----------------------------------");
            Console.WriteLine();
            Console.WriteLine("Fazendo a alteração");

            try
            {
                  // Criando a Conexão com o Servidor e o Banco de dados
                  strCmm = "UPDATE Categories set CategoryName='"+Nome+"',"
                        +"Description='"+Descricao+"' where CategoryID="+Id;

                  Console.WriteLine("String de Alteração Criada:");
                  Console.WriteLine(strCmm);
                  Console.WriteLine();

                  Console.WriteLine("Fazendo a conexão com o banco e dados");
                  conn = new SqlConnection(strCnx);
                  conn.Open();

                  Console.WriteLine("Criando um comando de alteração em SQL");
                  comm = new SqlCommand(strCmm,conn);

                  Console.WriteLine("Executando o comando de alteração");
                  comm.ExecuteNonQuery();
                  Console.WriteLine();
                  Console.WriteLine("Alterado com sucesso");
                  Console.WriteLine();

                  conn.Close();
            }
            catch (Exception ex)
            {
                  Console.WriteLine();
                  Console.WriteLine("ERROR");
                  Console.WriteLine("-----------------");
                  Console.WriteLine("Erro ao tentar alterar");
                  Console.WriteLine(ex.ToString());
                  Console.WriteLine("-----------------");
            }
      }
}
```

Excluindo Dados

Agora iremos usa o comando *Delete* do SQL para realizar a exclusão de um registro.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Incluir
{
   public static void Main()
   {
      // Criando a variáveis para alteração da tabela CATEGORIAS
      string Id;

      string strCnx = "Server=wxp-01;Initial Catalog=Northwind;user id=sa";
      string strCmm;
      SqlConnection conn;
      SqlCommand comm;

      // Listando as categorias
      try
      {
         strCmm = "select CategoryID,CategoryName,Description from Categories";
         conn = new SqlConnection(strCnx);
         conn.Open();

         comm = new SqlCommand(strCmm, conn);

         SqlDataReader dr = comm.ExecuteReader();
         Console.WriteLine();
         Console.WriteLine("L I S T A G E M");
         Console.WriteLine("---------------");

         while (dr.Read())
         {
            Console.Write("[{0}] [{1}] [{2}]",dr[0],dr[1],dr[2]);
            Console.WriteLine();
         }

         Console.WriteLine("---------------");


         conn.Close();

      }
      catch (Exception e)
      {
         Console.WriteLine();
         Console.WriteLine("ERROR");
         Console.WriteLine("-----------------");
         Console.WriteLine("Erro ao tentar listar");
         Console.WriteLine(e.ToString());
         Console.WriteLine("-----------------");
      }

      Console.Write("Digite o CategoryID para a exclusão: ");
      Id = Console.ReadLine();
      Console.WriteLine();
      Console.WriteLine("Iniciando a Exclusão");

      try
      {
         // Criando a Conexão com o Servidor e o Banco de dados
```

```
            strCmm = "DELETE FROM Categories where CategoryID="+Id;

            Console.WriteLine("String de Exclusão Criada:");
            Console.WriteLine(strCmm);
            Console.WriteLine();

            Console.WriteLine("Fazendo a conexão com o banco e dados");
            conn = new SqlConnection(strCnx);
            conn.Open();

            Console.WriteLine("Criando um comando de exclusão em SQL");
            comm = new SqlCommand(strCmm,conn);

            Console.WriteLine("Executando o comando de exclusão");
            comm.ExecuteNonQuery();
            Console.WriteLine();
            Console.WriteLine("Excluído com sucesso");
            Console.WriteLine();

            conn.Close();
        }
        catch (Exception ex)
        {
            Console.WriteLine();
            Console.WriteLine("ERROR");
            Console.WriteLine("-----------------");
            Console.WriteLine("Erro ao tentar excluir");
            Console.WriteLine(ex.ToString());
            Console.WriteLine("-----------------");
        }
    }
}
```

Local do Arquivo no CD: CD-ROM:\curso/O Database/e_excluir.cs