Banco do Estado do Rio Grande do Sul

Treinamento em C#

Unidade de Desenvolvimento de Sistemas

Autor: Maurício Teichmann Ritter
Editores: Felipe Chaves Rodrigues e Nelio Maraschin
Data do Documento: 09/04/2009
Atualizacão: 04/12/2013

Sumário

1. Construindo aplicações em C#

Processo de construção e compilação

Um programador C# pode escolher dentre diversas ferramentas para a criação de seus programas, desde um simples programa para edição de textos até uma ferramenta completa com uma série de recursos como o Visual Studio.NET 2005 (IDE Integrada). Além destas ferramentas, existem ainda o Visual C# Express e o SharpDevelop, que são ferramentas gratuitas disponíveis na internet.

O processo de construção de um programa C# nada mais é do que a edição de um programa através de um editor de textos e a compilação deste programa através de um programa.

O compilador da linguagem C# acompanha o .NET Framework, e pode ser encontrado na pasta de instalação do framework (em geral C:\WINDOWS\Microsoft.NET\Framework\v<versão do framework>).

O .NET Framework SDK

Para construirmos aplicações em .NET, uma ferramenta fundamental que deve ser instalada em nossa estação de trabalho é o .NET Framework Sdk. Este pacote de ferramentas prepara a máquina não só para permitir a execução de aplicações .NET como também permite o desenvolvimento de tais aplicações. O SDK acompanha a maioria das ferramentas de desenvolvimento (como o Visual Studio.NET), mas existe a opção de não utilizar uma IDE de desenvolvimento. Sua instalação é totalmente recomendada.

Uma série de ferramentas acompanha o .NET Framework Sdk. Essas ferramentas permitem a execução de uma série de tarefas relacionadas ao desenvolvimento de aplicações, como a importação de web services, a geração de classes, a visualização de DLLs, etc.

O .NET Framework SDK encontra-se em geral na pasta C:\Arquivos de programas\Microsoft Visual Studio 8\SDK\v2.0. Nesta pasta você encontra um arquivo StartHere.HTM que é considerado o ponto de entrada para a documentação do SDK do .NET Framework.

O Visual Studio.NET 2005

É lógico que a utilização do bloco de notas e do compilador de linha de comando torna o desenvolvimento das aplicações extremamente complexo. A prática mais comum é utilizarmos uma ferramenta de desenvolvimento, como por exemplo o Visual Studio.NET.

O Microsoft Visual Studio 2005 é um editor de código para .NET que permite tanto a criação de aplicações cliente servidor e componentes, como também inclui a ferramenta de desenvolvimento para Web, que possui um conjunto de ferramentas e utilitários para criação de sites da Web ASP.NET versão 2.0. O Visual Studio 2005 representa uma melhoria evolutiva no suporte para a criação de sites Web, trazendo os benefícios de produtividade do ambiente de desenvolvimento integrado (IDE) ao apresentar uma enorme gama de melhorias.

Editor de código e interface

O Visual Studio .NET traz um editor de interface para páginas ASP.NET muito familiar com o editor de interface para o Visual Basic, a intenção é que os desenvolvedores Visual Basic se sintam à vontade para criar Web Forms como se estivessem criando Windows Forms.

É possível criar página apenas arrastando controles da caixa de ferramentas para dentro da página, assim como qualquer editor WYSIWYG, a grande diferença é que, como nos editores Windows Forms, clicando duas vezes sobre um controle, o Visual Studio automaticamente cria um manipulador para um evento daquele controle.

Também é possível criar páginas ASP.NET utilizando a linguagem HTML, basta apenas alterar a forma de visualização da página, as funcionalidades fornecidas pelo editor visual são as mesmas, ou seja, também é possível arrastar controles para dentro do HTML e alterar suas propriedades utilizando a caixa de propriedades. Além disso o Visual Studio .NET irá fornecer um ambiente de codificação HTML muito rico e poderoso.

Como falado anteriormente, para criar uma aplicação em ASP.NET o desenvolvedor utilizará uma das linguagens disponíveis no .NET Framework, e utilizará todos os seus recursos, sem exceções, portanto o editor de código envolvendo lógica da página, será o mesmo utilizado para construir Windows Forms.

Desta forma, mesmo que o desenvolvedor não saiba a linguagem HTML, ele estará apto a criar páginas Web Forms sem problemas, já que ele terá um editor de interface tão poderoso quanto o do Windows Forms e poderá criar a lógica da aplicação exatamente como cria a lógica em formulários Windows Forms. Outro detalhe importante, é que mesmo sem ele nunca ter acompanhado o problema de compatibilidade entre navegadores, ele pode criar a página sem problemas, porque quem vai se preocupar com isso por ele é o ASP.NET.

Web server integrado

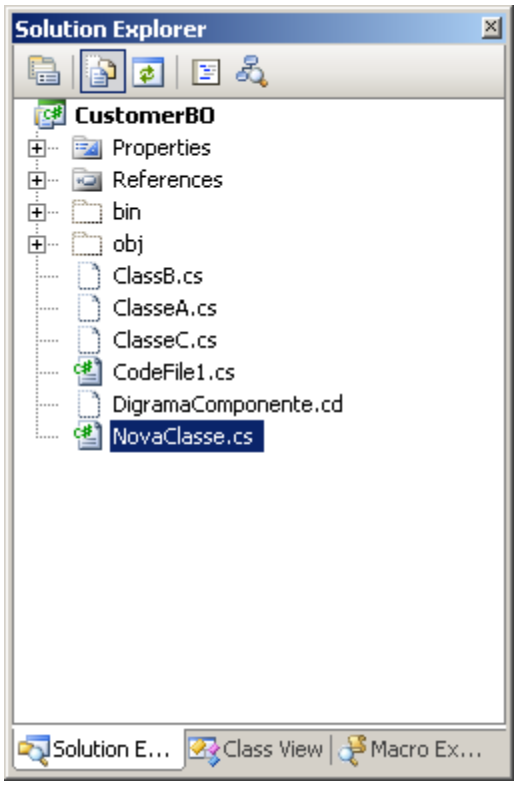
O Visual Studio 2005 traz integrado a seu sistema, um web server totalmente desvinculado ao IIS. Agora, aqueles que não têm IIS instalado em seus computadores, podem utilizar o Visual Studio 2005 para criar seus sites e web services e executá-los facilmente.

O web server integrado, é seguro pois não permite acesso por parte de outras máquinas (somente a máquina que esta rodando o Visual Studio pode acessá-lo), e possui acesso através de portas randômicas.

Solution explorer

O solution explorer é a janela onde conseguimos visualizar todos os projetos componentes de uma solução criada no Visual Studio.NET. Para cada projeto, podemos ainda abrir os arquivos componentes da solução e adicionar arquivos novos.

Através do botão “Show All Files” ainda é possível visualizar todos os arquivos que compõem a pasta do projeto, independente de fazerem parte do projeto ou não.



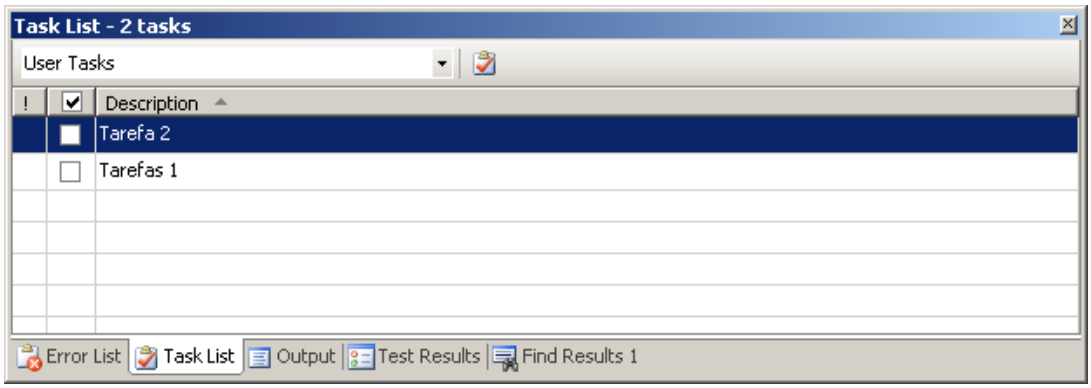
Task list

Permite manipular as tarefas geradas pelo usuário e pelo Visual Studio .NET. As tarefas podem ser adicionadas manualmente a janela de lista de tarefas ou podem ser adicionais através de tokens de programação inseridos no código.

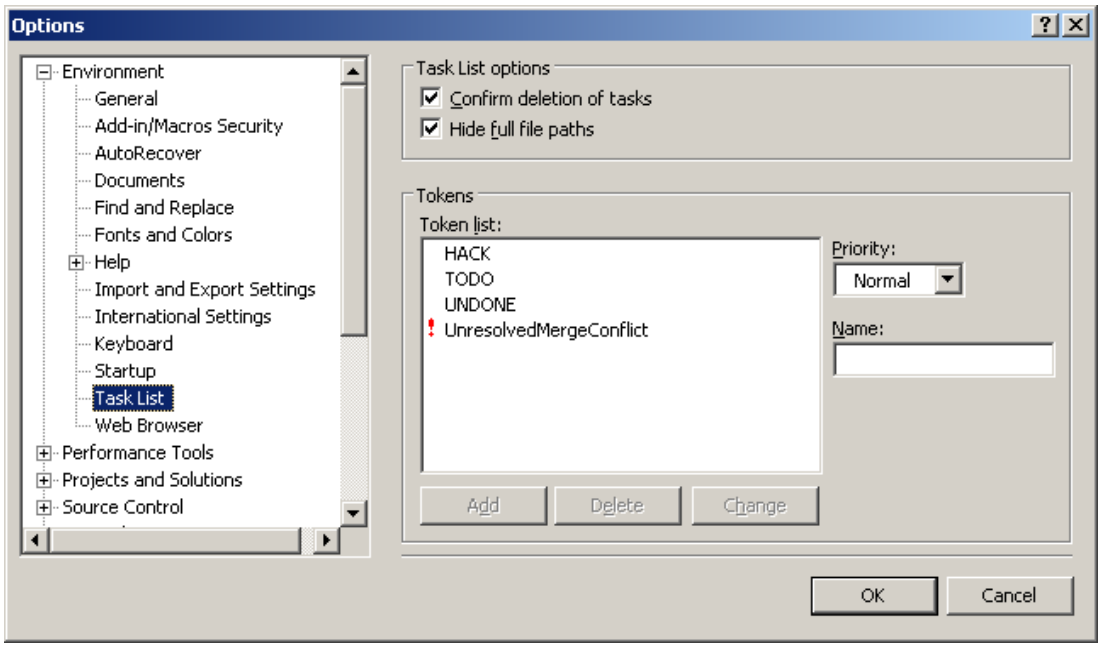
Para inserirmos algum item falante na programação, podemos incluir por exemplo um comentário com a marca TODO, como o exemplo a seguir.

//TODO: Implementar o código para validação do usuário

O comentário acima geraria uma nova tarefa na lista de tarefas do Visual Studio.



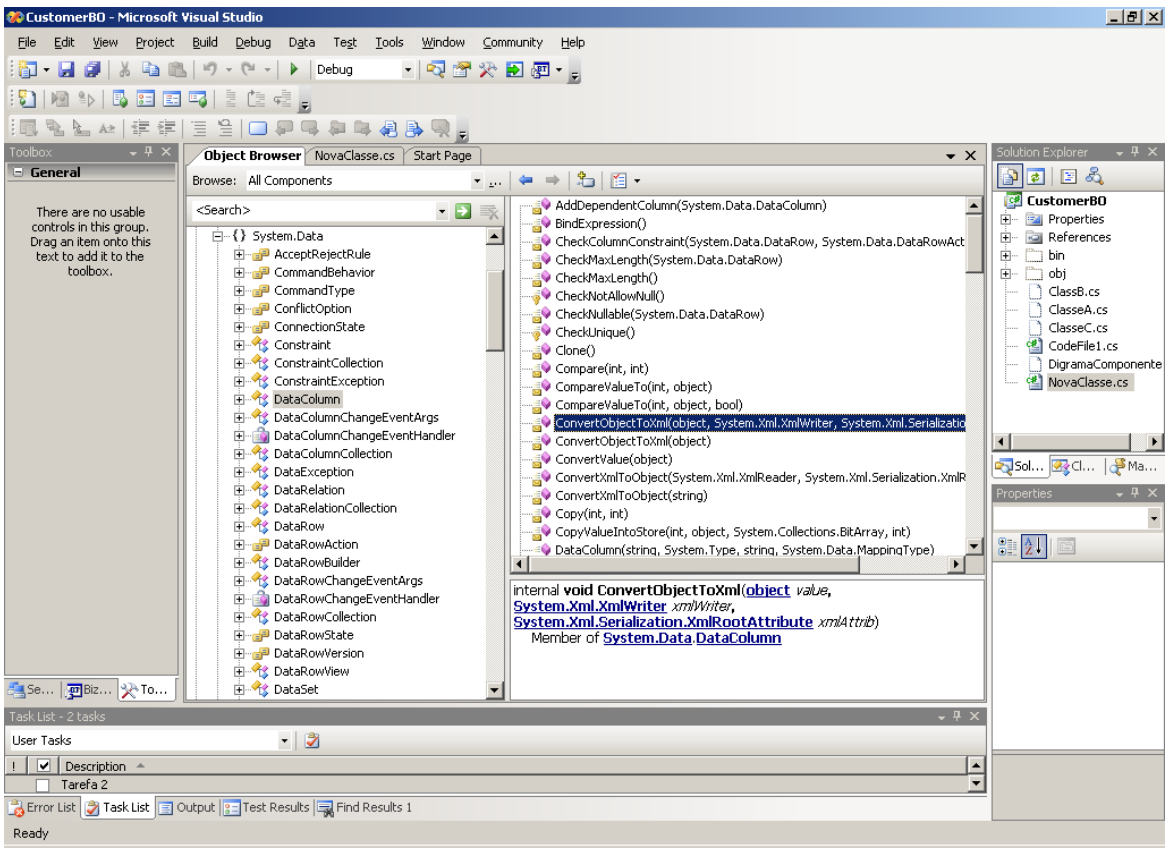
Os tokens utilizados para a criação de tarefas através de comentários podem ser configurados através do menu Tools, opção Options, na área “Environment”/”Task List”.



A lista de tarefas ainda apresenta os erros de compilação quando compilamos um programa em qualquer linguagem disponível no Visual Studio.

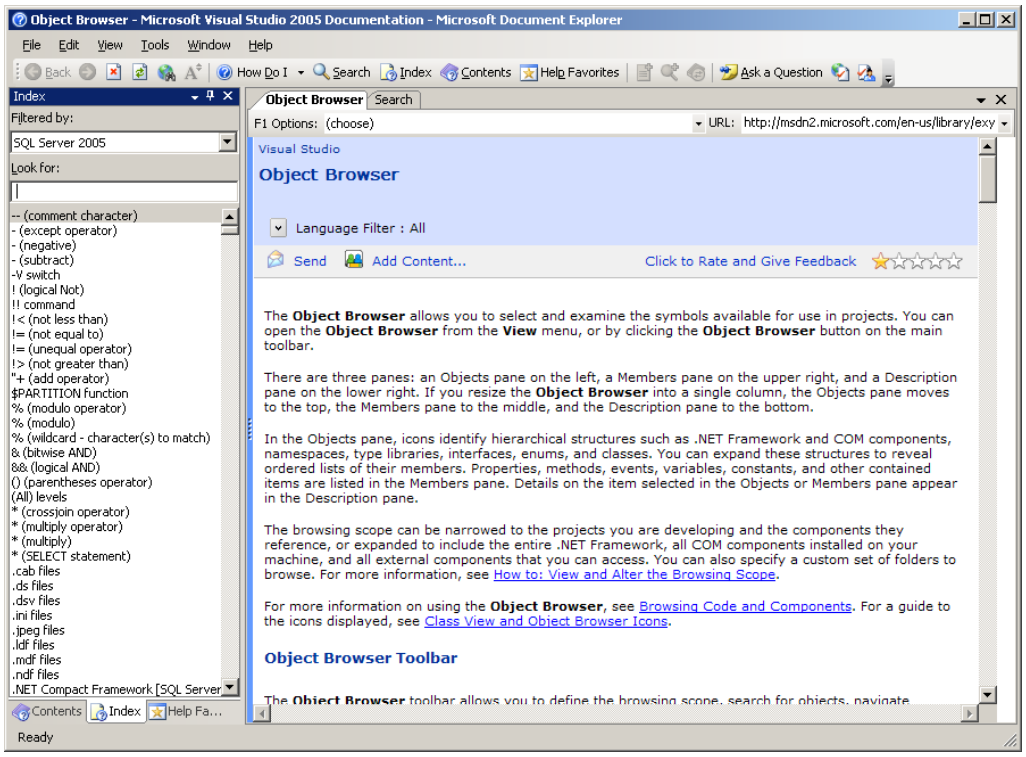
Object browser

A janela object browser fornece informações sobre os diferentes objetos do .NET Framework e dos projetos, como suas propriedades, eventos e métodos. Esta janela pode ser acessada através do menu View, na opção “Object Browser”



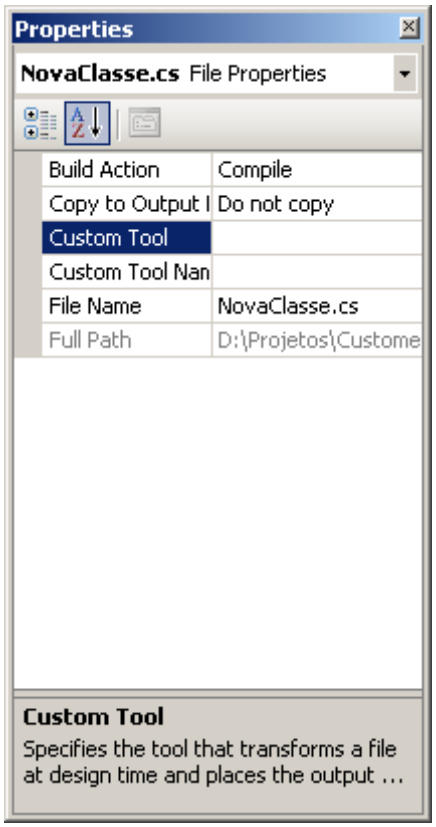
Dynamic help

O Visual Studio ainda disponibiliza o recurso de help dinâmico, que pode ser acessado a partir de qualquer janela da ferramenta. Para ativarmos o help dinâmico, basta pressionarmos a tecla F1 que a ferramenta disponibilizará um help contextualizado com a função ativa no momento.



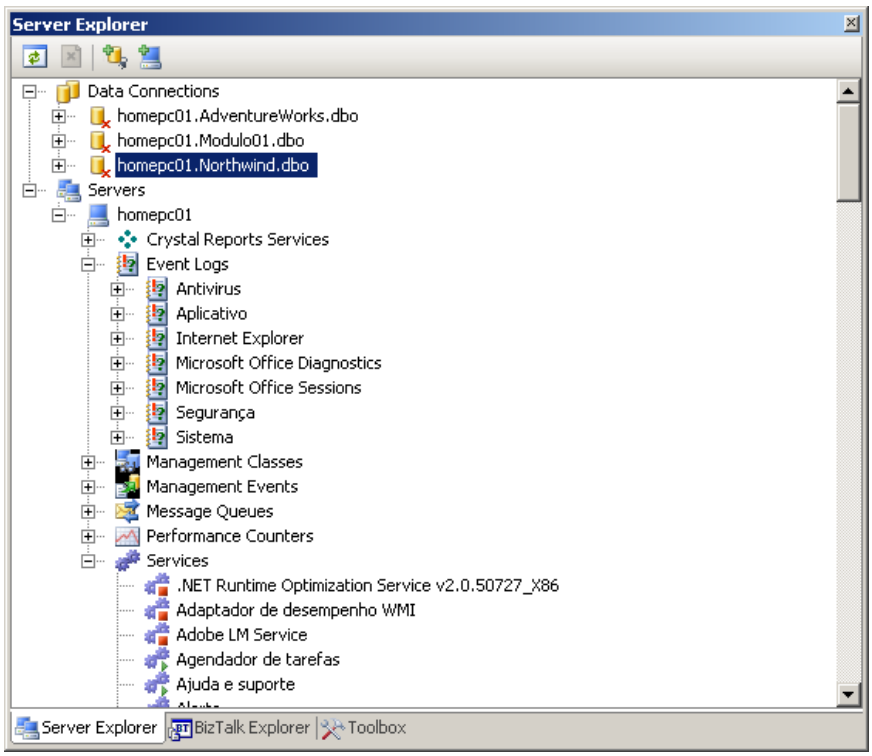
Properties

A janela de Properties disponibiliza as propriedades dos objetos quando estamos trabalhando com componentes que fazem uso de uma área de design. Para acessarmos esta janela, devemos estar em modo de design. A janela pode ser acessada através da tecla F4.



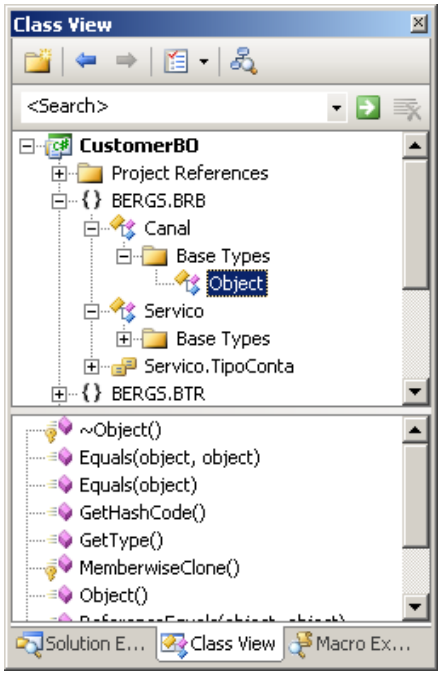
Server explorer

O Server explorer permite o acesso a recursos de servidores que podem ser utilizados durante o desenvolvimento da aplicação. Através desta janela é possível iniciar e parar serviços de um servidor, ver os bancos de dados disponíveis, a log de eventos e outros recursos adicionais. Também é possível a criação de conexões com banco de dados para manipulação de objetos como tabelas, procedures e views.



Class view

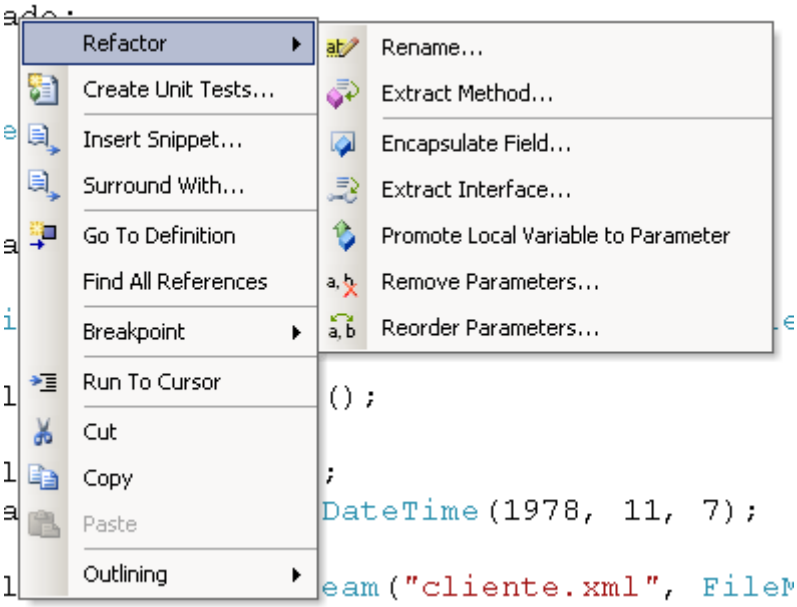
A janela Class View apresenta uma visualização dos tipos definidos no seu projeto de forma hierárquica (namespaces, classes, heranças, etc.). Um quadro de combinação (combo box) permite a busca por tipos de modo rápido e simples. A parte inferior apresenta os membros do tipo selecionado na parte superior, permitindo uma navegação simples até a definição do membro no código.



Refactoring

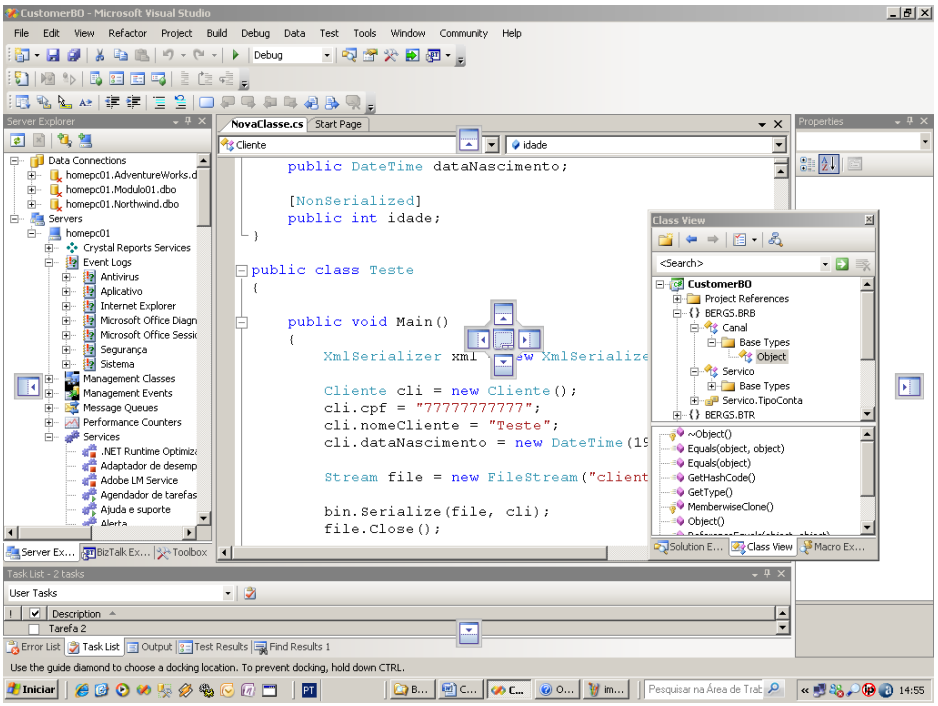
Além das janelas já apresentadas aqui, o Visual Studio ainda possui uma série de recursos que

facilitam a edição e manutenção do código fonte dos programas. Este recurso também é conhecido por refactoring, e permite a troca de nome de métodos ou de variáveis e a replicação desta mudança em todo o código fonte, permite a extração de interfaces de classes dentre outras funcionalidades.



Alinhamento de janelas

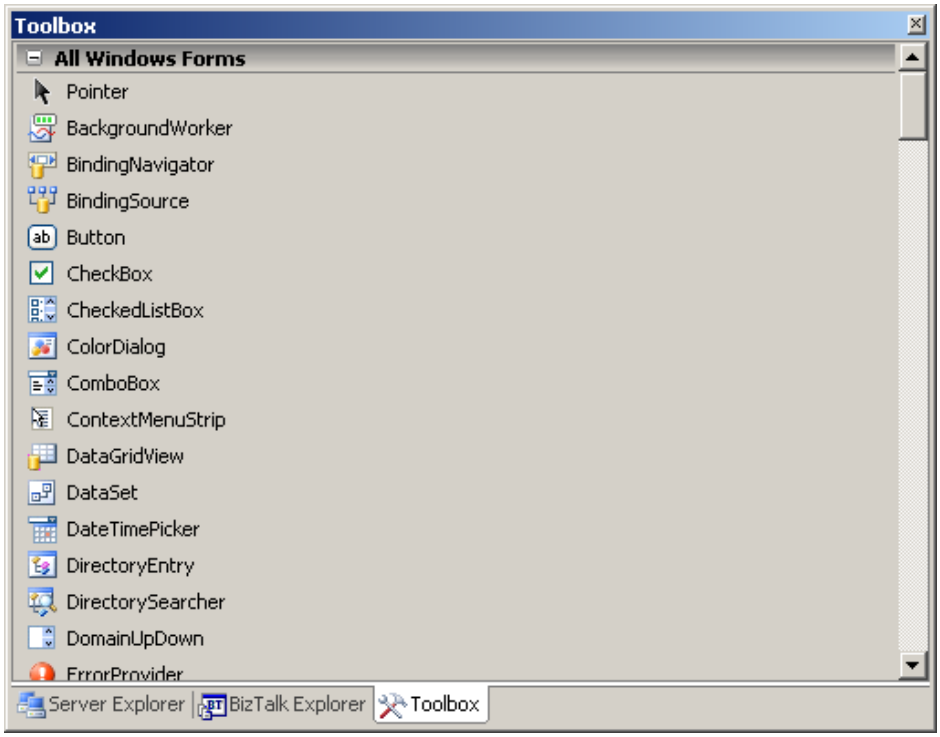
O Visual Studio também apresenta um novo sistema para posicionamento das janelas (Solution Explorer, Properties e etc.) dentro ad IDE. Agora, quando arrastamos uma janela para fora da estrutura padrão, fica muito simples arranjar o layout da maneira escolhida. Para isso, basta que o desenvolvedor clique com o botão direito do mouse sobre a barra da Janela e selecione a opção Docktable. Ao arrastar a janela pela IDE, algumas marcas inteligentes são mostradas, permitindo que o desenvolvedor enquadre a janela onde quiser como visto abaixo.



Notem as marcas inteligentes - Elas permitem que a janela seja alinha a partir da janela sobreposta; Ao clicar no centro a janela será alinhada ao centro, enquanto os direcionais permitem que seja alinha acima, abaixo, a esquerda ou a direita da janela sobreposta

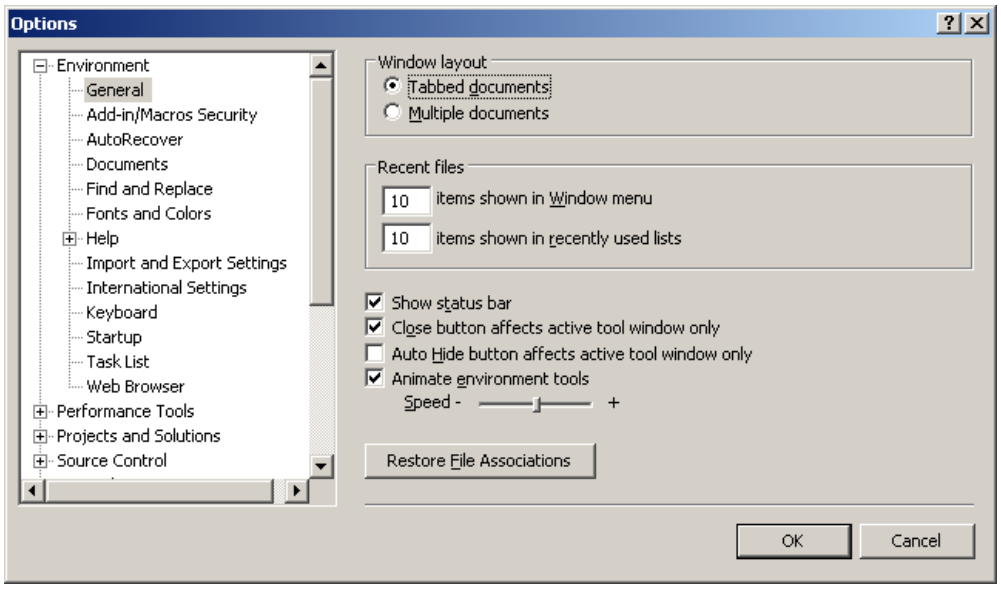
ToolBox

A toolbox é a janela que traz os controles para o uso no designer do Visual Studio. O Visual Studio oferece uma série de controles que facilitam o desenvolvimento tanto de aplicações web como aplicações com formulários.

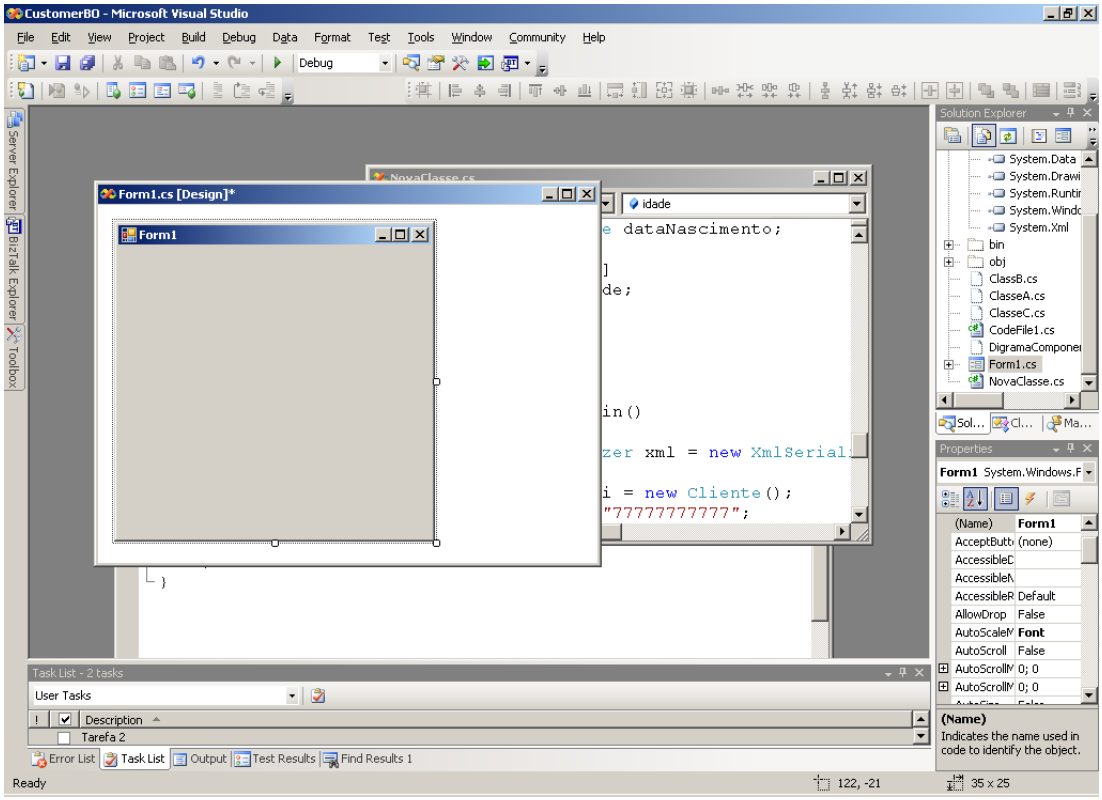


Sistema de exibição de páginas

A IDE do Visual Studio pode ser alterada para que os arquivos abertos sejam mostrados seguindo o padrão MDI environment, que utiliza o método de transformar todos os arquivos abertos e janelas de documentos, como no Microsoft Word e demais produtos da Microsoft. Para habilitar este modo, basta acessar o menu Tools -> Options, e logo em seguida acessar o item Environment/General marcando a opção Multiple documents nas opções do lado direito, como visto abaixo:



O modelo Mutiple Documents é semelhante ao modelo utilizado pela versão anterior do Visual Basic.

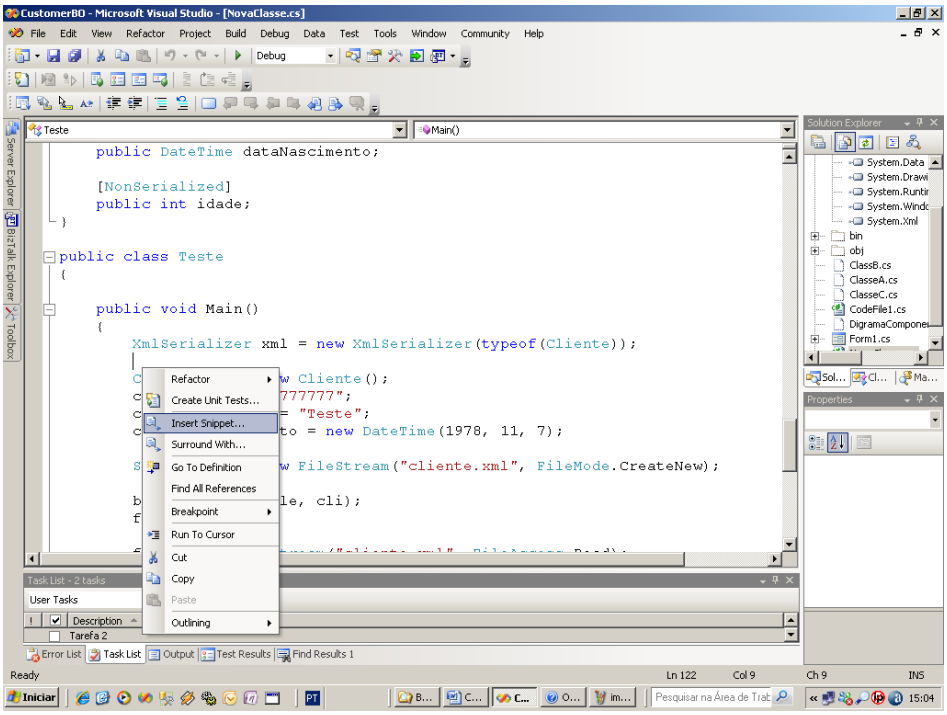


Os documentos abertos são exibidos como janelas flutuantes, permitindo ao desenvolvedor posicioná-las de acordo com suas necessidades.

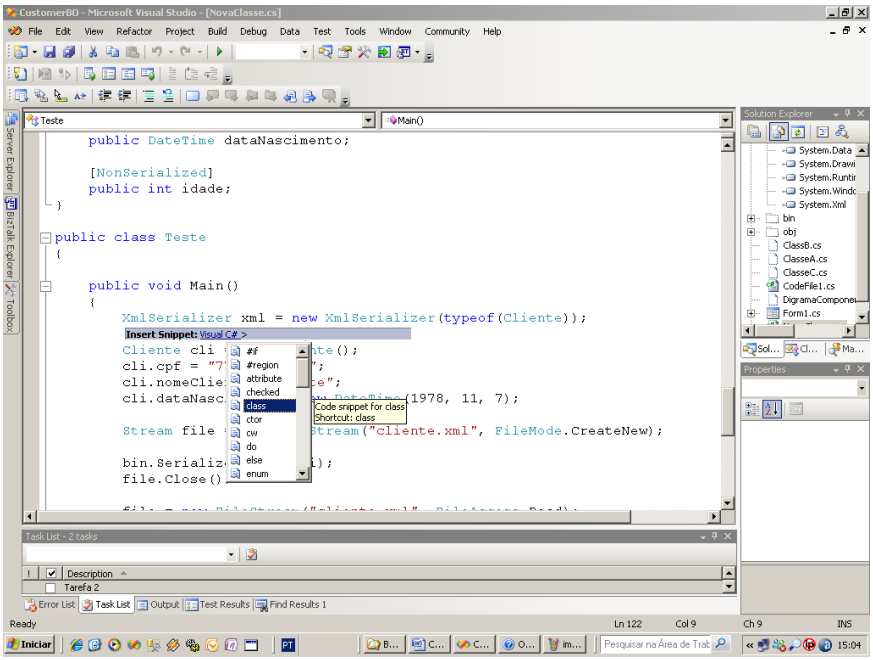
Code Snippets

Outra funcionalidade interessante são os code snippets, que auxiliam a criação de códigos muito redundantes. Quem já não se cansou de ter que escrever várias vezes o mesmo código para criar um DataSet? Como essa funcionalidade, basta clicarmos com o botão da direita em nosso

Code Editor e escolher a opção *Insert Snippet*. Será exibida uma janela com os tipos de códigos que podem ser exibidos. Logo em seguida escolhemos, por exemplo, a opção *Acessing Data* e por fim escolhemos a opção *Fill a DataSet with the Result of a Parameterized Stored Procedures.* O código resultante pode ser visto abaixo.



Uma janela é aberta para que um tipo de código seja escolhido pelo desenvolvedor.



O desenvolvedor poderá escolher o tipo de código que será inserido no Code Editor.

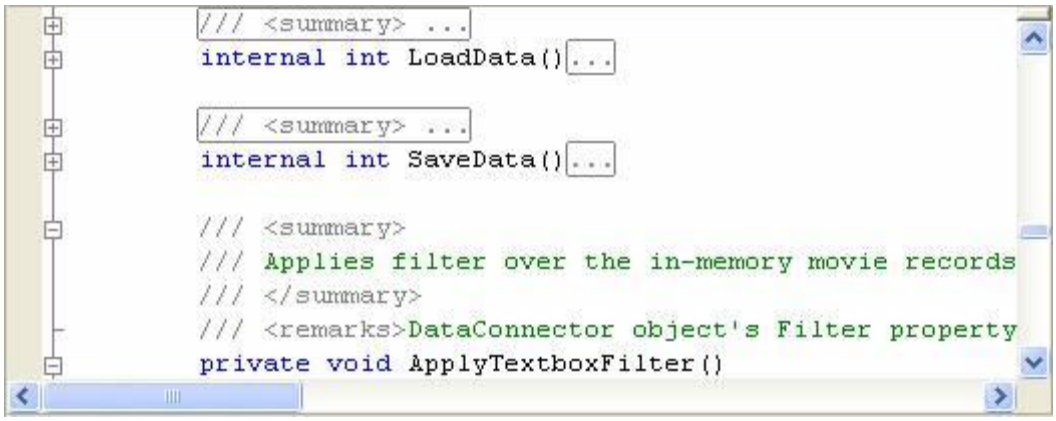
Por fim, o código é gerado automaticamente para que o desenvolvedor apenas modifique as informações básicas, como nome da Stored Procedure (no caso do exemplo), e informe os parâmetros para execução.

IntelliSense

O intellisence fornece um quadro de listagem com opções sensíveis ao contexto para completar o código. Além disto, durante a navegação da lista, dicas são apresentadas com descrições de cada item.

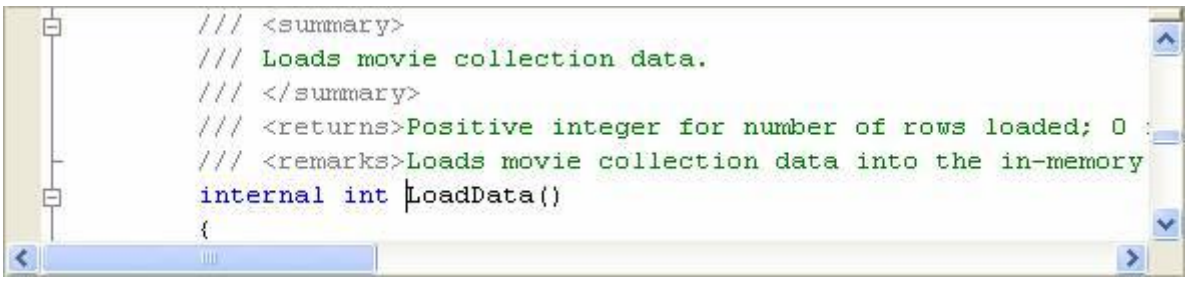
Regiões de ocultação

Permitem a visualização somente da parte do código que está sendo manipulada num determinando contexto.



Comentários XML

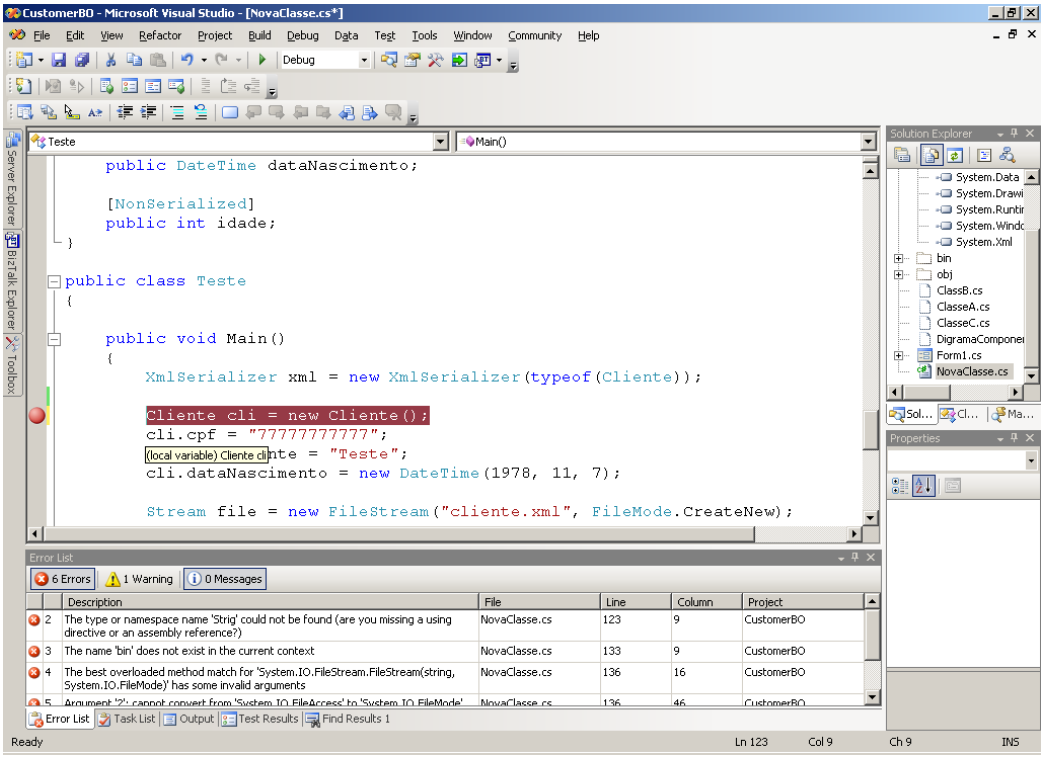
Os comentários em XML permitem a documentação do código com um conjunto de marcadores (tags) que podem ser utilizados dependendo do contexto. Estes comentários de documentação devem preceder imediatamente um tipo definido pelo usuário (como uma classe, uma estrutura, uma interface ou um delegate) ou um membro (como um campo, uma propriedade, um método ou um evento). O IDE auxilia na documentação preenchendo automaticamente com marcadores XML comuns de acordo com o contexto do código abaixo do local de inclusão dos comentários de documentação (///).



Breakpoints

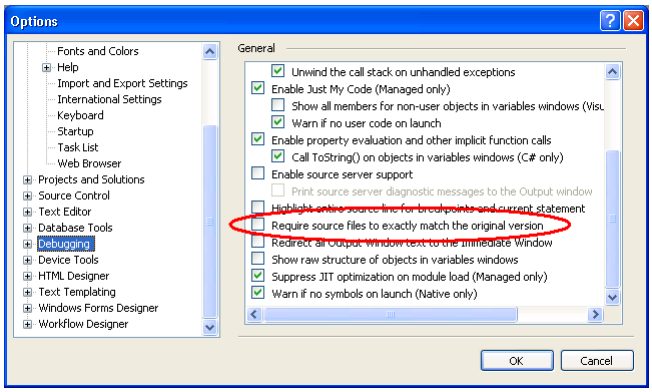
Os breakpoints definem linhas de parada para execução passo a passo em modo de depuração

para análise de código.



Debug

Ao executar o passo-a-passo dos programas, podem ocorrer erro indicando que a versão do fonte que estamos rodando não corresponde ao objeto compilado, nesses casos, devemos marcar a opção indicada abaixo na imagem.



2. Fundamentos da linguagem C#

Estrutura de um programa C# Simples

A linguagem C# exige que todo o programa esteja contido dentro de uma definição de um tipo (ou seja, uma classe, interface, estrutura, enumeração, etc.). Diferente do Visual Basic e de outras linguagens, não é possível criarmos funções ou variáveis globais. Um programa simples escrito em C# pode ser visto no exemplo a seguir.

```
//By convention, C# files end with a *.cs file extension.
using System;
class HelloClass
{
        public static int Main()
        {
             Console.WriteLine("Hello World!");
              Console.ReadLine();
              return 0;
        }
}
```

No programa acima temos a definição da classe HelloClass que suporte um único método com o nome de Main. Todo o executável de um programa C# deve obrigatoriamente possuir um função Main, que funciona como ponto de entrada para o programa.

Você pode perceber que a função Main possui as palavras “static” e “public” na sua declaração. Entraremos em detalhes sobre estes dois comandos mais a diante. Para este momento, o importante é entendermos que estes dois comandos indicam que esta função é visível e pode ser executada sem a necessidade de criação de um instancia de classe.

Como você pode notar, o delimitador utilizado para a definição da classe C# são os caracteres “{“ e “}”. Diferente de outras linguagens que utilizando comandos para delimitar blocos de código (como END no caso do cobol e End Function, End Sub no caso do VB) o C# utiliza para todo o bloco de código os caracteres de chaves.

Além desta característica, é importante salientar que o C# é uma linguagem que faz distinção entre letras maiúsculas e minúsculas. Todas as palavras reservadas do C# são escritas em letras minúsculas, portanto, escrever Public ao invés de public pode gerar um erro de compilação do programa.

Outro ponto interessante a ser observado é o uso da classe Console. A classe console apresenta uma série de funções que são utilizadas para a interação com a interface de linha de comando. Note que ao utilizar o método WriteLine conseguimos imprimir uma informação qualquer na tela do prompt de comando.

Da mesma forma, o comando ReadLine serve para obter alguma informação do usuário que está interagindo com o programa. Neste caso estamos utilizando este comando apenas para que a janela de prompt não seja fechada após a execução dos comandos.

Observe também que antes da definição de nossa classe HelloClass temos um comando using. Este comando é utilizado para indicar o uso de uma biblioteca dentro do nosso programa. A biblioteca System é a biblioteca que contém os tipos e classes mais primitivos do C#, dentre eles a classe console que permite a interação com o prompt de comando.

Utilizando a classe System.Environment

Além da classe console, ainda temos disponível a classe environment. Esta classe nos permite obter uma série de informações a respeito do ambiente de execução de nossos programas. Da mesma forma que a classe Console, a classe Environment faz parte da biblioteca System, por tanto não é necessário utilizarmos a notação System.Environment.

Abaixo temos o exemplo de um outro programa que faz uso da classe Console e da Classe environment.

```
using System;
namespace Modulo01
{
   class Program
   {
      static void Main()
      {
         Console.WriteLine("Programa de testes do C#.");
         Console.Write("Diretório Atual: ");
         Console.WriteLine(Environment.CurrentDirectory);
         Console.Write("Nome da máquina: ");
         Console.WriteLine(Environment.MachineName);
         Console.Write("Versão do Sistema Operacional: ");
         Console.WriteLine(Environment.OSVersion);
         Console.Write("Usuário logado na máquina: ");
         Console.WriteLine(Environment.UserName);

         Console.ReadLine();
      }
   }
}
```

No programa acima fazemos o uso novamente da classe Console, utilizando os métodos Write e WriteLine. O método Write exibe o conteúdo na tela sem uma quebra de linha, já o WriteLine exibe o conteúdo na tela com uma quebra de linha.

Através da classe Environment podemos obter uma série de informações sobre o ambiente de execução de nosso programa, dentre elas o diretório atual da aplicação, o nome da máquina do usuário, a versão do sistema operacional e o usuário logado na máquina.

Declarando variáveis

A declaração de variáveis no C# não possui um comando específico como em outras linguagens de programação (Dim no Visual Basic e var no javascript). Ao contrário destes outros programas, a declaração do C# é feita informando-se o tipo de dado a ser utilizado e em seguida o nome da variável, conforme o exemplo a seguir.

```
int numero;
string nome;
```

Os tipos de dados disponíveis no C# possuem uma correspondência no Common Type System do NET Framework. A tabela a seguir mostra todos os tipos disponíveis.

Common Type System Type	***C#***	***Visual Basic.NET***
System.Byte	byte	Byte

Common Type System Type	*C#*	*Visual Basic.NET*
System.SByte	sbyte	SByte
System.Int16	short	Short
System.Int32	int	Integer
System.Int64	long	Long
System.UInt16	ushort	UShort
System.UInt32	uint	UInteger
System.UInt64	ulong	ULong
System.Single	single	Single
System.Double	double	Double
System.Object	object	Object
System.Char	char	Char
System.String	string	String
System.Decimal	decimal	Decimal
System.Boolean	bool	Boolean

Note que cada tipo do C# possui uma correspondência a um tipo do Common Type System (CTS), portanto utilizar o nome do CTS ou a palavra chave do C# é a mesma coisa.

```
Int32 numero;
```

... é o mesmo que...

```
int numero;


string nome;
```

... é o mesmo que...

```
String nome;
```

Atribuição de valores e inicialização de variáveis

Uma variável pode ser inicializada na sua declaração, bastando para isto informar o sinal de atribuição (=) logo após o nome da variável, juntamente com o valor a ser atribuído.

```
int contador = 0;
string nome = "Maurício Ritter";
```

Além disso, é possível atribuir valores as variáveis em qualquer ponto do código utilizando o sinal de igualdade. Não é necessário informar o tipo na frente do nome da variável, pois isto caracterizaria uma segunda declaração da mesma variável. Um exemplo de atribuição de valores pode ser observado a seguir.

```
using System;
namespace Modulo01
{
   class Program
   {
      static void Main()
      {
         string usuario;
         usuario = Environment.UserName;

         string mensagem = "";
         mensagem = "Bem vindo ao .NET " + usuario;

         int valor;
         valor = 10;

         Int32 contador = 0;
      }
   }
}
```

As variáveis podem ser declaradas dentro de métodos (como a função Main) e também dentro da própria classe (classe Program). Não é possível declarar variávies FORA de um MÉTODO ou CLASSE. No C# não existem variáveis "globais".

Utilizando operadores

Após a declaração das variáveis, é possível utilizarmos operadores para modificarmos o valor destas variáveis. Os operadores utilizam os símbolos aritméticos padrão, como soma, subtração e divisão.

O exemplo a seguir mostra a utilização de alguns operadores aritméticos.

```
using System;

namespace Modulo02
{
   class Program
   {
      static void Main()
      {
         string usuario;
         usuario = Environment.UserName;

         string mensagem = "";
         mensagem = "Bem vindo ao .NET " + usuario;
         mensagem += ", agora são " + DateTime.Now.ToString("t");
         Console.WriteLine(mensagem);

         int valor1, valor2, valor3;
         valor1 = 10;
         valor2 = 20;
         valor3 = valor1 + valor2;
         valor3 += 5;
         valor3 -= 8;
         valor3 *= 9;
         valor3++;

         Console.ReadLine();

      }
   }
}
```

No que o operador de soma pode ser utilizado com tipos de dado strings, o que representa uma operação de concatenação.

Os operadores disponíveis para utilização no .NET são:

Operador	Função
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo, devolve o resto.
+=	Valor da variável mais o operando
-=	Valor da variável menos o operando
*=	Valor da variável vezes o operando
/=	Valor da variável divido pelo operando

Conversão de dados

Quando precisamos atribuir o valor de uma variável para outra variável de um tipo diferente o C# exige a utilização de uma função de conversão. Diferente de outras linguagens, como o Visual Basic, o C# exige a conversão explicita de dados na maioria dos casos. Vamos analisar os diferentes tipos de conversão possíveis no C#.

Conversão explícita

A conversão explícita pode ser feita através da utilização de operadores de "cast" (conversão). Um operador de cast é utilizado informando-se o tipo de dados entre parênteses antes da atribuição do valor a variável de um tipo diferente. Um exemplo de operação de cast explícito pode ser observada a seguir.

```
int valorInteiro;
double valorGrande = 10;
valorInteiro = (int)valorGrande;
```

No exemplo acima vemos para converter um valor do tipo double para o tipo inteiro, é necessário informarmos um operador de conversão antes da atribuição (int). Se não informarmos a função de conversão, conforme o exemplo acima, receberemos um erro de compilação.

Conversão implícita

Apesar da exigência de uma função de conversão na maioria dos casos, conforme vimos no exemplo acima, existem alguns casos em que a função de conversão não se faz necessária. Quando fazemos a conversão de um tipo de dados de menor magnitude para um de maior magnitude, a

função de conversão não é necessária, tendo em vista que não corremos o risco de ter perda de valores no processo de conversão, conforme o exemplo a seguir.

```
int valorInteiro = 10;
double valorGrande;
// Conversão implícita
valorGrande = valorInteiro;
```

Conversões para strings

A conversão de strings funciona de uma maneira um pouco diferente do que as conversões apresentadas nos exemplos acima. Os strings só podem ser convertidos através de funções especiais, capazes de realizar este tipo de conversão.

Convertendo valores para string

A conversão de uma variável qualquer para string é o processo mais simples dentro de uma aplicação .NET. Para convertermos um valor para o tipo string, basta informarmos a chamada do método ToString() após o nome da variável que o possui o valor a ser convertido. Todo e qualquer tipo de dados do .NET possuí a função ToString e em alguns casos esta função pode inclusive receber parâmetros. Um exemplo de conversão pode para o tipo string pode ser observado a seguir.

```
using System;

namespace Modulo02
{
   class Program
   {
      static void Main()
      {
         string mensagem;

         int valor1, valor2;
         valor1 = 10;
         valor2 = 15;

         int result = valor1 + valor2;

         mensagem = "O resultado da soma de 10 + 15 é "
            + result.ToString();
      }
   }
}
```

No exemplo acima estamos somando duas variáveis do tipo inteiro e concatenando o valor convertido para string em uma variável chamada mensagem.

Convertendo string para outros valores

A conversão de strings para outros valores se dá através do uso de uma classe especial para conversão, a classe Convert. A classe Convert é capaz de converter uma string para qualquer tipo primitivo de dados. A função a ser utilizada deve ser To<Tipo de Dado>. Abaixo temos um exemplo de uso da classe Convert.

```
string numero = "1000";

int   valor  = Convert.ToInt32(numero);
short valor2 = Convert.ToInt16(numero);
```

```
DateTime dt = Convert.ToDateTime("01/01/2007");
```

Outra opção para conversão de strings é a utilização da função Parse de um tipo específico. A maioria dos tipos de dados suporta a função Parse, que permite a conversão de um string para um valor, conforme exemplo a seguir.

```
int   valor  = Int32.Parse(numero);
short valor2 = Int16.Parse(numero);

DateTime dt = DateTime.Parse("01/01/2007");
```

[Exercício 2.1 da lista de exercícios]

Definindo classes e criando objetos

Agora que você tem uma visão geral da função Main, da declaração de variáveis e das funções de conversão, vamos trabalhar com a construção de objetos.

O termo objeto refere-se simplesmente a uma instância de uma classe em memória. Podemos declarar variáveis capazes de armazenar objetos da mesma forma que variáveis de tipos primitivos, bastando para isto informar o tipo da classe e o nome da variável. É importante salientar que a simples criação de uma variável de um tipo de classe não representa um objeto no C#, portanto, o exemplo a seguir representa um trecho de código inválido.

```
class HelloClass
{
   public void DisplayMessage()
   {
      Console.WriteLine("Hello da classe!");
   }

}

class Program
{
   static void Main()
   {
      HelloClass objeto;

      // ERRO ! Objeto não está instanciado
      objeto.DisplayMessage();
   }
}
```

Para criarmos um objeto, devemos utilizar o comando new. Este comando é responsável por alocar os recursos necessários para a utilização desta classe (recursos como memória, por exemplo). A variável funciona apenas como uma referência para este local onde os recursos estão alocados.

```
HelloClass objeto1 = new HelloClass();

HelloClass objeto2;
objeto2 = new HelloClass();

objeto1.DisplayMessage();
objeto2.DisplayMessage();
```

No exemplo acima, cada variável objetoN contém um objeto da classe HelloClass.

Value Type e Reference Types

É importante salientar que existe uma grande diferença entre as variáveis de tipos primitivos e as variáveis que contém as classes. As variáveis de tipos primitivos (int, double, etc.) possuem a sua própria cópia dos valores. Isto significa dizer que em qualquer operação de atribuição o que ocorre é uma cópia de um valor de uma variável para outra variável, conforme o exemplo a seguir.

```
int a, b;

a = 10;
b = a;
b = 5;
// a continua com 10
// a operação acima não afeta a
// outra variável, cada uma tem
// a sua cópia do valor;3
```

Quando estamos trabalhando com classes, as variáveis não contem um valor, mas sim uma referência para uma área de memória que contem o valor. Desta forma, se atribuímos uma variável para outra variável, estas passam a apontar para a mesma referência, e as alterações feitas em uma variável afetam os valores da outra variável, conforme o exemplo.

```
Classe a, b;

a = new Classe();
a.valor = 10;
b = a;
b.valor = 500;
// A e B apontam para
// a mesma referencia
// a alteração feita a partir
// da variável B, afeta a variável a
```

Criando métodos em uma classe

Diferente do Visual Basic o C# não faz diferença entre métodos e subrotinas. No C# os métodos e as sub-rotinas são declarados utilizando a mesma sintaxe.

Inicialmente é preciso informar o tipo de visibilidade do método. Diferente do Visual Basic onde a visibilidade Default é pública, no C# a visibilidade padrão é private.

Após informarmos a visibilidade do método é preciso informar o seu tipo de retorno. Caso o método não tenha nenhum retorno (equivalente às sub-rotinas do Visual Basic) deverá ser utilizada a palavra reservada void.

Em seguida deve ser informado, entre parênteses, o conjunto de parâmetros aceitos pelo método. Os parâmetros são declarados da mesma forma que as variáveis (tipo de dado seguido do nome do parâmetro) e devem ser separados por vírgula. Após a declaração dos parâmetros devemos então informar o corpo do método entre chaves. Para métodos que retornam algum valor é obrigatório informar a cláusula return dentro do corpo do método.

A seguir podemos observar alguns exemplos de métodos.

```
public void InsereRegistro(string nomeCliente,
    string CNPJ,
    string endereco,
    int numero,
    int complemento)
```

```
{
    // Aqui não temos clásula return
    // pois o método não retorna nada
}

public int Soma(int v1, int v2)
{
    // O return neste caso é obrigatório
    return v1 + v2;
}
```

O código cima apresenta dois métodos, um com retorno e outro sem retorno.

Os modificadores in/out/ref

Quando criamos parâmetros em métodos do C#, estes parâmetros são passados por padrão como valores, ou seja, se estes valores forem alterados durante a execução do método, a mudança destes valores não será refletida na variável passada por parâmetro. Este comportamento é atribuído ao modificador de parâmetro in, que é implícito e não precisa ser informado. O exemplo abaixo mostra a passagem de parâmetros por valor.

```
public void AlteraValores(int valor)
{
    valor += 10;
}

public void Main()
{
    int valor = 100;
    AlteraValores(valor);

    // Valor continua 100, não é alterado
    System.Console.WriteLine(valor.ToString());
}
```

Este comportamento pode ser alterado utilizado os moficadores ref e out.

O modificador ref

O parâmetro ref permite a passagem de valores por referência. Utilizando a passagem de valores por referência, os valores modificados dentro da função refletem suas alterações para a função chamadora. Para utilizarmos o operador ref, devemos informá-lo tanto na declaração do parâmetro como também na chamada do método.

```
public void AlteraValores(ref int valor)
{
    valor += 10;
}

public void Main()
{
    int valor = 100;
    AlteraValores(ref valor);

    // Valor fica com 110
    System.Console.WriteLine(valor.ToString());
}
```

O modificador out

A diferença entre o modificador out e o ref é que o out permite a passagem de uma variável

não inicializada por parâmetro, o que não é permitido em parâmetros ref. Veja o exemplo a seguir.

```
public void AlteraValores(out int valor)
{
   valor = 0;
   valor += 10;
}

public void Main()
{
   int valor;
   AlteraValores(out valor);

   // Valor recebe 10
   System.Console.WriteLine(valor.ToString());
}
```

Quando utilizamos o modificador out, a variável deve obrigatoriamente ser inicializada dentro da função que contem o parâmetro out.

Criando propriedades

As propriedades refletem o estado de uma classe dentro. Nas propriedades devemos colocar valores que representem informações armazenas pelas classes. As propriedades não são simplesmente variáveis com visibilidade pública, mas sim estruturas de dados capazes de abstrair a complexidade de certa informação armazenada pela classe. Podemos por exemplo criar uma classe Tempo que exponha uma propriedade Horas e uma propriedade Minutos, mas internamente armazene os valores apenas em minutos.

Uma propriedade possui sempre dois métodos de acesso, um para gravação de valores e outro para leitura.

No exemplo a seguir podemos observar uma propriedade que armazena as informações sobre o trabalho de um funcionário.

```
class Trabalho
{
   private int m_Horas;

   public int HorasTrabalhadas
   {
      get
      {
         return m_Horas;
      }
      set
      {
         m_Horas = value;
      }
   }
}
```

No exemplo acima podemos observar que a declaração da propriedade é muito semelhante a de um método, com a diferença que não temos os parênteses utilizados para declaração de parâmetros.

No corpo da propriedade podemos observar os métodos de acesso get e set. O método get representa a ação de obtenção do valor da propriedade, já o método set representa uma atribuição de

um valor à propriedade. O método set utiliza a palavra reservada value para atribuir um novo valor a propriedade.

As propriedades armazenam internamente seus valores em variáveis privadas. Desta forma, podemos abstrair do usuário a complexidade de armazenamento das horas trabalhadas. Caso quiséssemos armazenas as horas trabalhadas em minutos, poderíamos simplesmente alterar os métodos de acesso da propriedade, sem prejudicar os usuários da propriedade da classe, conforme exemplo a seguir.

```
class Trabalho
{
   private int m_Minutos;

   public int HorasTrabalhadas
   {
      get
      {
         return (m_Minutos / 60);
      }
      set
      {
         m_Minutos = value * 60;
      }
   }
}
```

Utilizando construtores

As classes podem apresentar métodos especiais chamados de construtores. Os construtores são métodos especiais capazes de inicializar a classe com algum valor. O método construtor pode ser identificado por um método com o mesmo nome da classe. O exemplo a seguir mostra o exemplo de um construtor.

```
class Trabalho
{
   private int m_Minutos;

   public Trabalho()
   {
      // CONSTRUTOR
      // Inicializa minutos com zero
      m_Minutos = 0;
   }

   public int HorasTrabalhadas
   {
      get
      {
         return (m_Minutos / 60);
      }
      set
      {
         m_Minutos = value * 60;
      }
   }
}
```

O construtor é chamado sempre que instanciamos uma classe utilizando o comando new, conforme o exemplo a seguir.

```
Trabalho trab;
trab = new Trabalho(); //Construtor é chamado aqui
```

Também podemos fazer sobrecarga do construtor modificando os parâmetros. O exemplo a seguir mostra.

```
class Trabalho
{

   public Trabalho(int minutos)
   {
      // CONSTRUTOR
      // Inicializa minutos com o parâmetro recebido
      this.m_Minutos = minutos;
   }
```

Definindo constantes

Outra estrutura importante que pode ser construída no C# são as constantes. As constantes são valores que não alteram o seu valor durante a execução do programa. Em geral são utilizados para parâmetros de configuração ou para valores fixo representados por um nome para maior clareza.

As constantes são declaradas através da palavra const, e assim como qualquer outra estrutura de programação deve ser declarada dentro de uma classe.

```
const int MINUTOS_POR_HORA = 60;
```

[Exercício 2.2 da lista de exercícios]

Construindo Enumeradores

Os enumeradores são estruturas de dados que facilitam a leitura do código, permitindo definir constantes nomeadas para um conjunto de valores. Para definirmos um enum, utilizamos a seguinte sintaxe.

```
enum TipoConta
{
   Corrente,
   Poupanca,
   Investimento
}
```

No exemplo acima, temos um enumerador que permite o uso de três valores, Corrente, Poupança e Investimento. Os enumeradores são tipos de dados assim como as classes, o que nos permite a declaração de variáveis com o tipo do enum. Para atribuirmos valores a estas variáveis, utilizados o nome do enum, um ponto e o nome do valor nomeado dentro do enum, conforme o exemplo a seguir.

```
TipoConta tipo;
tipo = TipoConta.Corrente;
```

Os valores de um enum podem ser convertidos para strings ou para inteiros. Caso sejam convertidos para inteiros os valores são correspondentes à posição de declaração do valor nomeado no enum, começando por zero, conforme o exemplo a seguir.

```
TipoConta tipo;
tipo = TipoConta.Corrente;
// Zero neste caso
int tipoConta = System.Convert.ToInt32 (tipo);
// Será atribuida a string "Corrente"
string nomeTipo = tipo.ToString();
```

É possível atribuir valores inteiros a cada item do enum, bastando para isto informar o sinal de igual e o valor desejado na declaração do enum. Utilizando esta notação, o valor informado será utilizado quando ocorrer uma conversão do valor do enum para inteiro.

```
public enum TipoConta
{
   Corrente = 10,
   Poupanca = 20,
   Investimento = 30
}
```

[Exercício 2.3 da lista de exercícios]

Trabalhando com Arrays

Além das variáveis normais que criamos até o momento, o C# ainda possui suporte a criação de arrays. Os arrays são declarados de forma semelhante às variáveis convencionas, mas possuem um modificador [ ] que indica que a variável é um array, conforme exemplo.

```
// Apenas um inteiro
int valor;

// Um array de inteiros
int[] valores;
```

Quando declaramos um array no C#, este array encontra-se em um estado não inicializado. Para inicializar um array devemos utilizar o comando new indicando o tamanho a ser alocado para o array. O tamanho do array é informado dentro dos colchetes, conforme exemplo a seguir.

```
// Array com 5 inteiros
int[] valores = new int[5];

// Array com 10 nomes, de 0 à 9
string[] nomes = new string[10];
```

É possível ainda inicializar o array no seu dimensionamento, bastando para isto informar os itens do array logo após o comando new entre chaves, conforme o exemplo a seguir.

```
// Array com 5 inteiros
int[] valores = new int[5] { 1, 2, 3, 4, 5 };

// Sem informar a dimensão
string[] nomes = new string[] { "Nome1", "Nome2", "Nome3" };
```

Acessando elementos do array

Para acessarmos elementos dos arrays, devemos utilizar a sintaxe utilizando colchetes. Deve-se informar entre colchetes o índice do array a ser acessado. Os arrays sempre iniciam pelo índice 0, conforme o exemplo a seguir.

```
nomes[0] = "Nome1";
nomes[1] = "Nome2";

string var = nomes[3];
```

Redimensionando Arrays

O C# não possui um comando de redimensionamento de array, como o Visual Basic (comando Redim). Para redimensionarmos um array no C# devemos utilizar uma função da classe System.Array, a função Resize. Esta função possibilita o redimensionamento do array semelhante à função Redim do Visual Basic.

```
string[] nomes = new string[2];

nomes[0] = "Nome1";
nomes[1] = "Nome2";
System.Array.Resize<string>(ref nomes, 3);
nomes[2] = "Nome3";
```

Classes de coleções

Apesar de suportar array, as operações com este tipo de estrutura em geral são complexas de se implementar. O mais comum é utilizarmos as classes de coleção. Estas classes abstraem a complexidade dos array, tornando o redimensionamento e a obtenção de itens automáticos. As classes de coleção fazem parte da biblioteca (namespace) System.Collections. A seguir temos um exemplo da utilização da classe ArrayList.

```
System.Collections.ArrayList lista = new System.Collections.ArrayList();

lista.Add("Nome1");
lista.Add("Nome2");
lista.Add("Nome3");
lista.Remove("Nome2");
lista.RemoveAt(0);
```

É possível observar que utilizando o ArrayList não a necessidade de redimensionamento quando inserimos ou removemos um item. Nos próximos capítulos veremos mais detalhes sobre a utilização de lista utilizando Generics.

Comandos de decisão (IF e SWITCH)

Assim como qualquer outra linguagem de programação, o C# também possui comandos para implementação de critérios de decisão. O comando IF possui uma estrutura semelhante a outras linguagens de programação, mas diferente do visual basic, não requer o comando THEN.

Abaixo temos um exemplo de implementação do comando IF.

```
if (m_Saldo - valorSaque < 0)
{
   Console.WriteLine("Saldo insuficiente");
   return;
}
else
{
   m_Saldo -= valorSaque;
}
```

Como você pode observar a instrução de decisão deve ser colocada entre parênteses logo após o comando IF e não devemos informar a cláusula THEN. O bloco que delimita o comando IF é o padrão do C#, ou seja, as chaves. O comando ELSE é opcional e deve vir logo após o bloco de código contendo o código a ser executado caso a instrução seja verdadeira.

Dentro de um IF, podemos informar critérios compostos utilizando os operador E e OU. Ao contrário do Visual Basic, não utilizamos as palavras em inglês para representar estes operadores, mas sim os símbolos && para E e || (duplo pipe) para OU. O exemplo a seguir mostra uma lógica composta com estes operadores.

```
if (m_ContaEncerrada == false &&
   m_Saldo - valorSaque < 0)
{
   Console.WriteLine("Saldo insuficiente");
   return;
}
else
{
   m_Saldo -= valorSaque;
}
```

No exemplo acima podemos observar dois operadores no C#. Podemos observar o operador lógico && atuando como comando AND para os dois critérios. Também podemos observar o operador == representa o operador lógico de igualdade no C#. A seguir podemos observar a lista de operadores lógicos disponíveis.

Operador	Função
= =	Testa igualdade
!=	Testa não igualdade
>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual
!	Operador de negação
&&	Operador E
||	Operador OU

Muitas vezes precisamos testar uma determinada variável para um conjunto finito de valores, ou mesmo executar uma determinada instrução dependendo do valor contido em uma variável. O C# oferece o comando switch para estes casos, que funciona da mesma forma que o comando Select Case do Visual Basic. A sintaxe do comando switch pode ser observada a seguir.

```
switch (dia_semana)
{
   case 1:
      nome_dia = "domingo";
      break;
   case 2:
      nome_dia = "segunda-feira";
      break;
   case 3:
      nome_dia = "terça-feira";
      break;
   case 4:
      nome_dia = "quarta-feira";
      break;
```

```
        case 5:
            nome_dia = "quinta-feira";
            break;
        case 6:
            nome_dia = "sexta-feira";
            break;
        case 7:
            nome_dia = "sábado-feira";
            break;
        default:
            nome_dia =  "dia inválido";
            break;
    }
```

O comando switch avalia uma determinada expressão conforme um conjunto de valores, e executa um determinado conjunto de comandos para cada valor possível. No exemplo acima, estamos avaliando a variável inteira dia_semana. Para cada valor temos uma cláusula case dentro do bloco do comando switch.

Os blocos de case não são delimitados por chaves, mas sim por um comando break. Podemos inserir quantas linhas de código forem necessárias para um valor de case e ao final do conjunto de valores devemos informar a cláusula break;

A cláusula default é utilizada para os casos em que a variável não contenha nenhum valor encontrado em alguma das cláusulas case do comando.

O comando switch pode ser utilizado tanto para variáveis numéricas como também para variáveis do tipo string.

Comandos de Repetição (For/While/Foreach)

O C# possui três tipos de comando de repetição, for, foreach e while. Esses comandos são utilizados para repetir um determinado bloco de código a partir de um critério de decisão (com exceção do foreach que é utilizado para fazer iterações em uma lista).

O comando For

O comando de repetição mais simples é o comando for e possui a seguinte sintaxe:

for(variárivel = valorinicial; expressão booleana; expressão de incremento)

No exemplo a seguir vemos a utilização do comando for para executar um bloco de código um número determinado de vezes.

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

Note que a variável utilizada no for é declarada dentro do próprio comando. Utilizando o comando desta forma, a variável tem visibilidade somente dentro do for.

O comando while

O comando while pode ser utilizado de duas formas, sem o comando “do” e com o comando “do”.

Na primeira opção, informamos a cláusula while e a condição que será avaliada para verificar se o bloco de código contido no loop deve ser executado ou não. Caso a expressão retorne valor falso o bloco de código pode não ser executado.

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

A outra opção é utilizarmos o comando "do". Neste caso, a iteração é executada pelo menos uma vez e a expressão booleana é avaliada ao final do bloco de código.

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

O comando foreach

O comando foreach é utilizado nos casos em que queremos fazer uma iteração em uma lista de valores ou um array. Para cada elemento na coleção especificada, o comando executará o bloco de código especificado, conforme exemplo a seguir.

```
string[] arquivos = System.IO.Directory.GetFiles("C:\\WINDOWS");

foreach (string arquivo in arquivos)
{
    Console.WriteLine(arquivo);
}
```

Interrompendo a execução do loop

Outra opção disponível para o comando de loop é o comando para interrupção, o comando break. Através deste comando é possível interromper a execução de qualquer um dos loops antes que a expressão utilizada como condição seja falsa.

```
foreach (string arquivo in arquivos)
{
    Console.WriteLine(arquivo);
    if (System.IO.File.GetLastWriteTime(arquivo) > DateTime.Today)
        break;
}
```

Ainda temos a opção de interromper uma iteração específica do loop, passando para a próxima iteração.

```
foreach (string arquivo in arquivos)
{
    if (arquivo.EndsWith(".dll"))
        continue;

    // Só será executado para arquivos
    // com extensão diferente de .dll
    Console.WriteLine(arquivo);
}
```

[Exercício 2.4 da lista de exercícios]

Definindo e utilizando namespaces

Os namespaces são estruturas de programação do .NET que permitem a contextualização de estruturas de programação. Você pode entender os namespaces como sub-bibliotecas. Os namespaces são um agrupamento lógico das classes. Definimos os namespaces utilizando a cláusula "namespace". Dentro de um namespace podemos colocar classes, estruturas, enumerados, etc.

O exemplo a seguir mostra o uso de um namespace:

```
namespace Bergs.Brb
{
   public class Servico
   {
      //...
   }
   public class Canal
   {
      //...
   }
}
```

Quando uma classe faz parte de um namespace, o nome desta classe é composto pelo namespace o qual ela está contida mais o nome da classe (separado por pontos). Se temos uma classe chamada ContaCorrente, dentro de um namespace Bergs.Bcc, o nome real desta classe é Bergs.Bcc.ContaCorrente (este nome completo também é chamado de nome totalmente qualificado).

Quando queremos referenciar uma classe podemos utilizar tanto o seu nome totalmente qualificado (com namespace) ou simplesmente o nome da classe. Para utilizarmos somente o nome da classe, devemos informar no topo do arquivo de código uma cláusula using indicando o uso do namespace que contém a classe. É importante salientar que o uso da cláusula using NÃO é OBRIGATÓRIO, ele apenas facilita a codificação, pois permite a utilização do nome da classe ao invés dos nomes totalmente qualificados.

Veja o exemplo a seguir.

```
namespace Bergs.Pxc
{
   public class Teste
   {
      public void subrotina()
      {
         BERGS.BRB.Servico obj = new BERGS.BRB.Servico();
      }
   }
}
```

Neste exemplo temos o uso do objeto utilizando o nome totalmente qualificado. No próximo exemplo, temos o uso do objeto utilizando apenas o nome da classe e a cláusula using.

```
using Bergs.Brb;

namespace Bergs.Pxc
{
   public class Teste
   {
      public void subrotina()
      {
         Servico obj = new Servico();
      }
```

```
    }
}
```

Trabalhando com métodos estáticos

Nas classes que trabalhamos até agora, foi necessária a criação de uma instancia para podermos chamar métodos ou mesmo para atribuirmos propriedades.

O C# possui um tipo especial de método chamado método estático. Os métodos estáticos são métodos criados em uma classe o qual não dependem de uma instância de uma classe para funcionar. Os métodos estáticos são métodos que permitem a criação de funções genéricas, a partir da própria classe, sem a criação de uma instância.

Um bom exemplo de método estático são os métodos da classe System.Convert que utilizamos para conversão de dados. Note que em nenhum dos casos em que utilizamos estas funções foi necessária a criação de uma instância da classe System.Convert.

Para criarmos um método estático, basta informarmos a palavra static na declaração do método, conforme o exemplo a seguir.

```
namespace Bergs.Btr
{
    public class GravacaoLog
    {
        public static void GravaLog(string dados)
        {
            //...
        }
    }
}
```

Se precisássemos chamar o método acima, não seria necessário criar uma instância da classe log, bastaria informarmos o nome da classe e a função.

```
Bergs.Btr.GravacaoLog.GravaLog("Teste de gravação de Log");

// ou
using Bergs.Btr;
GravacaoLog.GravaLog("Teste de gravação de log");
```

Utilizando Strings do C#

O tipo string apesar de se comportar como um value type é na verdade um reference type. Apesar disso, devemos considerar que estamos trabalhando com um value type quando estamos manipulando strings, tendo em vista que este tipo de dado foi programado para funcionar como tal.

A classe string possui uma série de métodos estáticos e não estáticos, que são utilizamos para formatação, concatenação, desmembramento, substring, etc.

Vamos analisar alguns destes métodos da classe string.

O método Substring

O método substring é um método não estático que permite pegarmos uma porção de uma string. Para utilizarmos o método substring, basta chamar o método a partir de uma variável string ou mesmo de uma string literal, conforme o exemplo.

```
string nome = "Testando da Silva";

string sobreNome = nome.Substring(12,5);
```

O método Substring aceita como parâmetro a posição inicial que queremos obter e quantos caracteres devem ser extraídos. Caso não seja informado o número de caracteres a ser extraído, a função retornará o restante da string a partir da posição inicial informada.

O método IndexOf

O método IndexOf é utilizado para localizar uma determinada palavra dentro da string. Este método retornará a posição da string desejada. Caso a string não seja encontrada, será retornado o valor -1.

```
string nome = "Testando da Silva";
int pos = nome.IndexOf("Silva");
//A partir do índice 5
int pos2 = nome.IndexOf("Silva", 5);
```

Conforme vimos no exemplo, é possível informar a posição inicial para busca como um parâmetro adicional.

Funções ToUpper e ToLower

As funções ToUpper e ToLower permitem colocar uma string em letra minúsculas ou maiúsculas, conforme o exemplo a seguir.

```
string nome = "Maurício";

nome = nome.ToUpper();
nome = nome.ToLower();
```

Funções StartsWith e EndsWith

As funções startswith e endswith verificam se a string começo ou termina com uma determinada palavra ou caracter. Estas funções retornam um valor booleano (true ou false).

```
string arquivo = "brbsglvw.dll";

if(arquivo.StartsWith("brb"))
    Console.WriteLine("É do sistema BRB!");

if(arquivo.EndsWith(".dll"))
    Console.WriteLine("É uma dll!");
```

Funções TrimStart, TrimEnd e Trim

As funções de Trim servem para remover espaços em branco das strings. A função TrimStart remove os espaços em branco do início da string, já a função TrimEnd remove os espaços em branco do final da string. A função Trim remove os espaços em branco do inicio e do fim da string.

```
string nome = "   MAURICIO    ";

nome = nome.TrimEnd();
nome = nome.TrimStart();

nome = nome.Trim();
```

Funções PadLeft e PadRight

As funções PadLeft e PadRight servem para preencher uma string a esquerda ou a direita com um caracter especificado.

Os exemplos a seguir mostra o uso das funções PadLeft e PadRight.

```
string nome = "Mauricio";

nome = nome.PadRight(10, ' ');
// "Mauricio  "
string nsu = "123";
nsu = nsu.PadLeft(6, '0');
// "000123"
```

Funções String.Join e String.Split

A função Split serve para quebrar uma string em um array de strings de acordo com um separador. Ao contrário da função split, a função Join concatena um array de string inserindo um separador.

```
string linha = "Teste, 10, 20, 10/06/2007";

string[] campos = linha.Split(',');

string linhaNova = String.Join(';', campos);
```

A Função String.Format

A função String.Format é uma das funções mais versáteis da classe string. Esta função permite a criação de uma string baseada em um padrão. Podemos colocar dentro do padrão da string de formatação alguns tokens que poderiam ser substituídos por variáveis passadas por parâmetro no comando format.

```
string teste = String.Format("Vou colocar o {0} aqui.", "parâmetro");
```

No exemplo acima, o padrão da string (primeiro parâmetro) contém um token {0}. Este token indica que este valor deve ser substituído pelo segundo parâmetro passado para a função (neste caso a palavra “parâmetro”).

```
sTeste = String.Format(
            "Formatação de string com {0} parâmetro. Agora são {1}. Valor numérico: {2}",
            1, Now(), 15.5);
```

No exemplo acima, temos o padrão da string com mais de um token, com números de 0 a 2. A função também recebe três parâmetros adicionais que correspondem aos valores que serão substituídos na string.

Além de informarmos os tokens, podemos informar regras de formatação que serão utilizadas de acordo com o tipo de dado passado por parâmetro, conforme o exemplo,

```
sTeste = String.Format("Custom Format: {0:dd/MM/yyyy HH:mm:ss}", dtHoje);
sTeste = String.Format("Formatação do número inteiro: {0:D}.", iNumero);
sTeste = String.Format("Formatação do número inteiro: {0:D4}.", iNumero);
// Com 2 decimais sempre
```

```
sTeste = String.Format("Formatação do número com decimais: {0:0.00}.", 15.4);
// Com até 2 decimais
sTeste = String.Format("Formatação do número com até 2 decimais: {0:0.##}.", 15.456);
// alinhado à direita com 1 decimal
sTeste = String.Format("Formatação do número com decimais: {0:10:0.0}.", 15.456);
// alinhado à esquerda com 3 decimais (espaços à esquerda)
sTeste = String.Format("Formatação do número com decimais: {0:-10:0.000}.", 15.456);
```

A classe StringBuilder

Apesar da versatilidade da classe string, operações com string podem consumir muitos recursos e causar perda de desempenho. Isto ocorre porque a cada operação o framework cria um novo buffer para a string resultante da operação, ao invés de aproveitar o buffer que está sendo utilizado. Isto é mais aparente em operações de concatenação.

Para não criar este tipo de problema, podemos utilizar a classe StringBuilder, do namespace System.Text. Esta classe permite a criação de strings aproveitando o buffer utilizado, o que torna o código mais eficiente.

No exemplo a seguir temos um trecho de código com uma operação feita de forma ineficiente e outra mais eficiente usando o stringBuilder, veja a diferença.

```
// Código ineficiente
string sql;
sql =  "SELECT * FROM Products";
sql += " WHERE CategoryId=1 ";
sql += "AND UnitPrice>10";

// Código Eficiente
StringBuilder sb = new StringBuilder();
sb.Append("SELECT * FROM Products");
sb.Append("WHERE CategoryId=1 ");
sb.Append("AND UnitPrice>10");
sql = sb.ToString();
```

[Exercício 2.5 da lista de exercícios]

3. Utilizando o namespace System.IO

Introdução

Muitas vezes em nossas aplicações precisamos trabalhar com arquivos texto ou mesmo arquivos de outros tipos (binários, csvs, etc.). Neste capitulo veremos como programar a leitura e a gravação de arquivos no C# e também como fazemos para criar pastas e realizar operações as operações de copia, exclusão e troca de pastas no sistema de arquivos.

O Namespace System.IO

No namespace System.IO podemos encontrar todas as classes responsáveis pela manipulação de arquivos e pastas. Boa parte das classes que encontramos neste namespace possui métodos estáticos, o que nos possibilita chamar estes métodos sem criarmos uma instância da classe em questão. Para que possamos trabalhar com o namespace System.IO, devemos utilizar a cláusula using para facilitar nosso trabalho.

Criando um arquivo texto

Tendo importando o namespace System.IO, vamos agora criar um arquivo texto onde escreveremos algumas informações como exemplo.

Existem várias maneiras de se abrir um arquivo texto no C#. Uma maneira muito simples é utilizando as classes StreamReader e StreamWriter. Estas classes permitem a gravação e a leitura de arquivos de uma maneira simples, fornecendo ainda métodos para a gravação e leitura de Strings nestes arquivos. Para utilizarmos estas classes, basta criarmos uma instância das mesmas informando em seu construtor o nome do arquivo a ser aberto.

Através dos métodos WriteLine e ReadLine, podemos gravar e ler informações dos arquivos texto de uma maneira bem simples. No exemplo abaixo, iremos abrir um arquivo texto para gravação e gravar algumas informações dentro dele.

```
        StreamWriter oFile = new StreamWriter("C:\\windows\\temp\\teste.txt");
//ou
        StreamWriter oFile = new StreamWriter(@"C:\windows\temp\teste.txt");

        oFile.WriteLine("TESTE Linha 1");
        oFile.WriteLine("TESTE Linha 2");
        oFile.WriteLine("TESTE Linha 3");
        oFile.WriteLine("TESTE Linha 4");
        oFile.Close();
```

Note que ao finalizarmos a gravação das informações no arquivo, utilizamos o método Close para fechar o Stream. É importante fecharmos o arquivo sempre que terminamos de realizar qualquer operação com arquivos, do contrário o arquivo ficará aberto.

No exemplo abaixo, iremos criar uma instancia da classe StreamReader para ler o arquivo que acabamos de criar. Utilizaremos o método ReadLine para ler as informações do arquivo.

```
StreamReader oFileRead = new StreamReader("C:\\windows\\temp\\teste.txt");
//ou
StreamReader oFileRead = new StreamReader(@"C:\windows\temp\teste.txt");

string sLinha;
while(oFileRead.Peek() > -1)
{
```

```
            sLinha = oFileRead.ReadLine();
            Console.WriteLine(sLinha);
    }
    oFileRead.Close();
```

Notem que utilizamos o método Peek da classe StreamReader. Este método nos informa o número de caracteres restantes existentes no arquivo. Caso não existam mais caracteres a serem lidos, o método Peek retorna o valor de -1.

Mas afinal, o que é um Stream?

Como podemos ver, para trabalharmos com arquivos no .NET Framework, precisamos trabalhar com as classes de leitura e gravação de Stream. Mas afinal o que são os Streams?

Os Streams são nada mais nada menos do que uma visão genérica de uma sequencia de bytes, ou seja, um stream representa um conjunto de bytes armazenados em sequência. Para lermos um stream, utilizamos as duas classes que vimos no exemplo anterior, StreamReader e StreamWriter.

O conceito de Stream é extramente importante, pois ele é extensamente utilizado no .NET Framework. Com ele, não realizamos apenas a leitura de arquivos, mas de qualquer sequência de bytes armazenada desta forma (inclusive sequencias de bytes armazenadas em memória, ou MemoryStream).

Adicionando informações ao arquivo

Se executarmos o exemplo de gravação de arquivos apresentado acima mais de uma vez, veremos que o arquivo é sobrescrito a cada execução. Para criarmos um StreamWriter capaz de realizar uma operação de inserção ao final de um arquivo (Append) basta informarmos um parâmetro adicional no construtor do mesmo. No exemplo abaixo, estamos passando o valor de True para o parâmetro de append, permitindo que as informações sejam adicionadas no arquivo.

```
StreamWriter oFile = new StreamWriter("C:\\teste.txt", true);

oFile.WriteLine("TESTE Linha 1");
oFile.WriteLine("TESTE Linha 2");
oFile.WriteLine("TESTE Linha 3");
oFile.WriteLine("TESTE Linha 4");
oFile.Close();
```

Lendo arquivos binários

Outra maneira de lermos e gravarmos arquivos é utilizando a leitura e a gravação binária. Neste tipo de gravação não iremos trabalhar com linhas como fazemos com arquivos texto, mas sim com buffers ou arrays de bytes. Utilizaremos esses arrays tanto para leitura como para gravação de arquivos. Este tipo de leitura e gravação é extremamente útil para casos em os arquivos que estamos trabalhando não sejam arquivos texto.

No exemplo a seguir, estamos utilizando os métodos Read e Write para ler e gravar um buffer de caracteres no arquivo.

```
StreamReader oFileReadBin = new StreamReader(@"C:\teste.txt");
char[]  buffer = new char[4];
oFileReadBin.Read(buffer, 0, 5);
oFileReadBin.Close();
```

```
StreamWriter oFileWriteBin = new StreamWriter("C:\\teste.txt", true);
oFileWriteBin.Write(buffer);
oFileWriteBin.Close();
```

Abrindo arquivos em modos diferentes

Note que em nenhum dos exemplos acima especificamos opções de compartilhamento de arquivos. Isto ocorre porque as classes de StreamReader e StreamWriter abrem com um modo de compartilhamento padrão (permitem acesso a leitura do arquivo mas não permitem gravação do arquivo por qualquer processo). Se quiser alterar estas opções ou mesmo alterarmos outras opções na abertura do arquivo, precisamos utilizar o método Open da classe File. Este método nos permite indicar o modo de abertura do arquivo (Criação, Append, Leitura), o tipo de operação a ser realizada (Gravação, Leitura ou ambas) bem como o modo de compartilhamento (Exclusivo, Exclusivo para Leitura, Exclusivo para Gravação, Compartilhado).

No exemplo abaixo, estamos abrindo um arquivo utilizando o método Open da classe File. Esta classe não retorna uma classe no tipo StreamReader ou StreamWriter, e sim uma classe FileStream. Se quisermos podemos criar um StreamWriter ou StreamReader a partir do FileStream, para utilizarmos as facilidades como as funções WriteLine e ReadLine destas classes.

```
FileStream oFileExclusive = File.Open(@"C:\\teste.txt",
          FileMode.Append, FileAccess.Write,
          FileShare.None);
StreamWriter oFEWriter = new StreamWriter(oFileExclusive);
oFEWriter.WriteLine("TESTE linha 1");
oFEWriter.Close();
oFileExclusive.Close();
```

Criando pastas no sistema de arquivos

Agora que já vimos como criar e ler arquivos, vamos criar uma nova pasta para colocar este arquivo que acabamos de criar com nosso programa de exemplo. No namespace System.IO existe uma classe chamada Directory. Esta classe possui métodos para trabalharmos com pastas do sistema operacional. Para criarmos uma novo pasta basta chamarmos o método CreateDirectory, conforme o exemplo abaixo.

```
Directory.CreateDirectory(@"C:\Subdir");
```

Movendo e Copiando arquivos

A tarefa de mover arquivos e copiar arquivos também é simples. Para isto, basta utilizar os métodos compartilhados da classe File. O método MoveFile é utilizado para mover arquivos, enquanto o método CopyFile é utilizado para copiar arquivos.

No exemplo abaixo podemos ver um exemplo da utilização dos dois comandos.

```
File.Copy(@"C:\teste.txt", @"C:\Subdir\teste2.txt");
File.Move(@"C:\teste.txt", @"C:\Subdir\Teste.txt");
```

4. Orientação a objetos no C#

Entendendo a estrutura de classes no C#

Conforme já vimos nos exercícios anteriores, o C# é uma linguagem totalmente orientada a objetos. Até agora vimos alguns conceitos mais voltados para a criação de classes, implementação de métodos e propriedades. Neste capítulo veremos como programar alguns conceitos avançados de orientação a objetos utilizando o C#.

A grande vantagem do uso de linguagens orientadas a objeto é a possibilidade de agrupamento de uma série de funcionalidades em um único UDT (User Defined Type), que pode ser uma classe, estrutura ou mesmo outras construções. Dentro deste conceito, podemos observar um conceito importante da orientação a objetos, o conceito de encapsulamento.

Encapsulamento

O encapsulamento é implementado na linguagem C# através da implementação de métodos e propriedades das classes. As classes podem encapsular uma série de tarefas complexas, expondo ao usuário funções mais simples, que permitem uma codificação mais fácil.

Abstração

Outro conceito importante é o conceito de abstração. A abstração é utilizada para expor ao usuário apenas o que é realmente importante, escondendo dele os detalhes de implementação ou mesmo alguma complexidade gerada pela classe.

Modificadores de acesso public e private

Uma maneira de implementarmos encapsulamento juntamente com o conceito de abstração é implementarmos métodos e propriedades com os modificadores de acesso correto. Colocando modificadores de acesso, podemos expor ao usuário somente o que é realmente importante. Vamos analisar o exemplo a seguir.

```
public class SimuladorCredito
{
    public double[] CalculaValorParcelas(double valorEmprestimo,
        double numeroParcelas,
        int tipoCliente,
        DateTime dataPagamentoPrimeiraParcela)
    {
        //...
    }

    private double CalculaTaxaJuros(int tipoCliente)
    {
        //...
    }
}
```

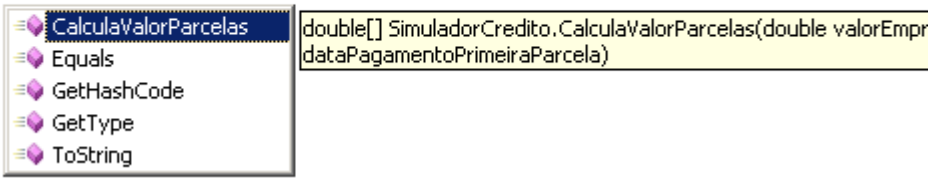
Na classe acima temos uma função pública chamada CalculaValorParcelas que recebe alguns parâmetros e devolve ao usuário um array de valores double contendo o valor das parcelas. Temos uma função adicional chamada CalculaTaxaJuros que pelo fato de ser uma função usada internamente pela classe para obter a taxa de juros baseada no tipo de cliente não é exposta ao

usuário da classe.

Se um usuário tentasse utilizar a classe, só poderia ver a função CalculaValorParcela, pois é a única com modificador de acesso public. Conforme exemplo abaixo.

```
SimuladorCredito sim = new SimuladorCredito();

sim.
```



Herança

A herança é um conceito extremamente utilizando dentro do .NET Framework. Este conceito é utilizado quando precisamos que uma determinada classe tenha todas as características de uma classe com algumas modificações em seu comportamento, ou mesmo algumas funcionalidades adicionais.

```
class ContaCorrente
{
   private double m_Saldo;
   private string m_NomeCorrentista;
   private int m_CodAgencia;
   private int m_CodConta;

   public string NomeCorrentista
   {
      get { return myVar; }
      set { myVar = value; }
   }
   public double Saldo
   {
   }
   public int Agencia
   {
   }
   public int Conta
   {
   }
   public double Sacar(double valor)
   {
   }
   public void Depositar(double valor)
   {
   }
}
```

A classe acima mostra a implementação de uma conta corrente, que permite o saque e o depósito de valores e contém algumas funcionalidades. Imagine que o sistema necessita suportar um novo tipo de conta. Este novo tipo de conta permitirá ao cliente ter um limite de crédito para saques. Além do limite de crédito, a conta deverá ter o valor da taxa de juros cobrada do cliente em caso de utilização do limite de crédito dentre outras funcionalidades. Este novo tipo de conta terá exatamente as mesmas características de uma conta corrente, mas deverá suportar estas novas

funcionalidades.

Ao invés de implementarmos uma nova classe, com uma cópia de todas as funcionalidades existentes na classe ContaCorrente, podemos criar uma nova classe e fazer com que ela herde as características da ContaCorrente. Desta forma, esta conta especial terá as mesmas funcionalidades da conta corrente, e algumas funcionalidades adicionais implementadas especificamente para esta classe. A vantagem de utilizarmos este tipo de implementação é que caso tenhamos que alterar alguma funcionalidade básica da conta corrente, estas alterações se propagaram automaticamente para a classe da conta especial.

Para implementarmos o mecanismo de herança no C#, basta informarmos o símbolo ":" após a declaração da classe e o nome da classe a qual queremos herdar as características. No exemplo a seguir temos a criação da classe ContaEspecial.

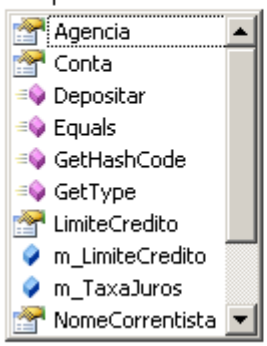
```
class ContaEspecial : ContaCorrente
{
   public double m_LimiteCredito;
   public Single m_TaxaJuros;

   public double LimiteCredito
   {
   }
}
```

Se tentarmos instanciar a classe ContaEspecial, poderemos observar que ela possui todos os métodos disponíveis na classe ContaCorrente e ainda a propriedade LimiteCredito, conforme o exemplo a seguir.

```
ContaEspecial cta = new ContaEspecial();

cta.
```



O modificador de acesso protected

As classes que herdam características de outras classes podem precisar alterar propriedades ou mesmo chamar métodos que estão disponíveis na classe pai. Isto é perfeitamente factível caso tenhamos estes métodos ou propriedades com visibilidade pública. Porém se esses métodos possuírem visibilidade private, eles poderão ser vistos apenas na classe pai e a classe filha não poderá ter acesso a estes valores.

Veja o exemplo a seguir.

```
class ContaCorrente
{
   private double m_Saldo;

   // ...
}

class ContaPoupanca : ContaCorrente
{
   public void AplicarReajuste(double rendimento)
   {
      // ERRO DE COMPILAÇÃO !!
      m_Saldo *= rendimento;
   }
}
```

No exemplo a seguir estamos tentando acessar a variável m_Saldo da classe pai em um método da classe filha. Como esta variável está declarada como private, este tipo de construção não é possível e gera um erro de compilação.

Para que os valores estejam visíveis na classe pai e também na classe filha, devemos declarar as variáveis na classe pai como protected, desta forma permitires o acesso desta variável em classes derivadas. Veja o exemplo agora implementado de forma correta.

```
class ContaCorrente
{
   protected double m_Saldo;

   // ...
}

class ContaPoupanca : ContaCorrente
{
   public void AplicarReajuste(double rendimento)
   {
      m_Saldo *= rendimento;
   }
}
```

Implementando polimorfismo

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.

Antes de verificarmos como o polimorfismo funciona sob o ponto de vista do usuário da classe, vamos ver como podemos implementar comportamentos diferentes para um mesmo método na classe pai e na classe filha.

O modificador virtual

No exemplo que utilizamos até aqui, temos a classe ContaCorrente e a classe ContaEspecial. Quando o usuário tenta efetuar um saque de algum valor e chama o método Sacar da classe conta especial, internamente ele estará chamando o método sacar da classe contacorrente, pois é onde o método está definido. Porém, nesta situação, o método sacar deveria ter um comportamento

diferente, pois deveria considerar o limite de crédito do usuário e não apenas o saldo em conta corrente. Para permitirmos que as classes filhas possam alterar o comporamento de funções da classe pai, reimplementando métodos, podemos utilizar o modificador virtual, conforme o exemplo a seguir.

```
class ContaCorrente
{
  protected double m_Saldo;
  private int nroConta;

  public int NroConta
  {
     get { return nroConta; }
  }

  public virtual double Sacar(double valor)
  {
     if (valor < m_Saldo)
        return 0;

     m_Saldo -= valor;
     return valor;
  }
  public ContaCorrente(int nro)
  {
     nroConta = nro;
  }
}
```

Declarando o método com este modificador, permitimos que ele seja reimplementado em classes que herdem suas características.

O modificador override

Agora vamos ver como podemos reimplementar o método Sacar na classe ContaEspecial. Para reimplementarmos o método, basta declararmos um novo método com a mesma assinatura do método da classe pai e com o modificador override, conforme o exemplo a seguir.

```
class ContaEspecial : ContaCorrente
{
   public double m_LimiteCredito;

   public override double Sacar(double valor)
   {
      if (valor < m_Saldo + m_LimiteCredito)
         return 0;

      m_Saldo -= valor;
      return valor;
   }
   public ContaEspecial(int nro) : base(nro)
   {
   }
}
```

A palavra reservada base

Se quisermos acessar a implementação original da classe pai, podemos utilizar a palavra reservada base para invocar o método da classe pai. Este modificar serve para acessar qualquer

membro da classe pai.

```
class ContaEspecial : ContaCorrente
{
   public double m_LimiteCredito;

   public override double Sacar(double valor)
   {
      if (base.Sacar(valor) > 0)
         return valor;
      if (valor > m_LimiteCredito + m_Saldo)
      {
         m_Saldo -= valor;
         return valor;
      }
      else
         return 0;

   }
}
```

Regras de conversão de dados

Agora que já vimos como funcionam os métodos polimórficos, vamos ver algumas regras relativas à conversão de dados de classes pais para classes filhas e como essa implementação leva ao conceito de polimorfismo.

Quando construímos uma instância de uma classe especializada (ex. ContaEspecial) podemos atribuir essa instância a uma variável do tipo da classe pai. Isto porque a classe especializada suporta todas as características da classe pai, e, portanto pode ser armazenada em uma variável deste tipo, conforme o exemplo a seguir.

```
ContaEspecial ctaA = new ContaEspecial();
ContaCorrente ctaB = cta;
ContaCorrente ctaC = new ContaEspecial();

ctaA.Sacar(1000);
ctaC.Sacar(1000);
```

Note que estamos atribuindo uma instancia da classe ContaEspecial para uma variável do tipo ContaCorrente. Mesmo estando armazenada em uma variável deste tipo, a instância continua sendo uma instância de conta especial.

O polimorfismo acontece justamente na chamada do método Sacar. Mesmo estando armazenada em uma variável ContaCorrente, a instância da classe ContaEspecial continua se comportando normalmente, ou seja, nas chamados do método Sacar a implementação definida na classe ContaEspecial continuará a ser chamada.

Caso tenhamos que fazer a conversão no caminho inverso, ou seja, da classe ContaCorrente para a classe ContaEspecial, temos que utilizar uma conversão explicita. Isto porque a variável conta corrente pode não conter uma instância da variável ContaEspecial, e por isso precisamos informar ao compilador que sabemos que a conversão pode ser efetuada, conforme exemplo a seguir.

```
ContaEspecial ctaA = new ContaEspecial();
ContaCorrente ctaB = cta;
ContaCorrente ctaC = new ContaEspecial();
```

```
ContaEspecial ctaD = (ContaEspecial)ctaB;
```

A classe System.Object

Um aspecto importante sobre as classes criadas no .NET Framework é que mesmo a classe não tendo uma herança explícita, esta classe herdará implicitamente as características da classe Syste.Object. A classe Object é a classe mais básica do .NET Framework e por este motivo, todo e qualquer objeto pode ser convertido para a classe object. A classe object é muito semelhante ao tipo Variant do Visual Basic. No exemplo a seguir temos uma série de instancias de classe sendo convertidas para o tipo object.

```
Cliente cli = new Cliente();
CustomerBO.Trabalho cust = new CustomerBO.Trabalho();

object obj = cli;
obj = cust;

cust = (CustomerBO.Trabalho)cust;
```

Note que os tipos também podem ser convertidos de object para outras classes, bastando para isto implementar um operador de conversão.

Podemos também converter variáveis value types para o tipo object. Como as variáveis value types não possuem uma referência aos valores, e sim o próprio valor, quando atribuímos uma variável value type para uma variável do tipo object estamos fazendo uma operação chamada boxing, ou seja, estamos colocando um valor value type dentro de um reference type. O mesmo acontece no processo inverso, quando colocamos uma variável object para um value type, estamos utilizando o processo de unboxing. O exemplo a seguir mostra como funcionam estas duas operações.

```
// boxing do número 10
// em um object
object obj = 10;

// unboxing da referência
// object para o valor inteiro
int valor = (int) obj;
```

Os métodos da classe Object

Assim como em qualquer outra classe pai, é possível reimplementar os métodos da classe Object. O procedimento mais comum é reimplementarmos o método ToString, que permite a conversão do tipo para string. A seguir temos a reimplementação de um tipo que não herda de nenhuma classe, mas herda implicitamente da classe Object. Neste caso, podemos reimplementar o método ToString, mesmo que não classe não haja qualquer indicação de um processo de herança.

```
public class Cliente
{
   public string nomeCliente;
   public string cpf;
   public DateTime dataNascimento;

   public int idade;
```

```
    public override string ToString()
    {
        string texto;
        texto = "Nome Cliente:" + nomeCliente;
        texto += ", CPF:" + cpf;
        texto += ", Data de Nascimento: " + dataNascimento;

        return texto;
    }
}
```

Documentação XML

Outro recurso disponível no C# é a documentação XML. Através desta documentação é possível especificarmos detalhes sobre um determinado método, classe ou propriedade.

A documentação XML é criada a partir de tags XML que são inseridas em blocos de comentários antes da declaração do método, propriedade ou classe. Para criarmos estes blocos de comentário, basta inserirmos um comentário com três barras (///) e utilizarmos as tags conforme necessário.

Implementando a documentação via XML, o usuário que utilizar sua classe poderá obter informações sobre os métodos através do intellisense. A seguir vemos um exemplo de implementação de documentação via XML.

```
/// <summary>
/// Aplica o reajuste no saldo da poupança de
/// acordo com o rendimento informado
/// </summary>
/// <param name="rendimento">
/// Deve ser informado em percentual (Ex.: 0.65)
/// </param>
public void AplicarReajuste(double rendimento)
{
    m_Saldo *= rendimento;
}
```

[Exercício 3.1 e 3.2 da lista de exercícios]

5. Ciclo de vida dos objetos no C#

Classes, objetos e referências

Nos capítulos anteriores verificamos alguns conceitos sobre a orientação a objetos e como podemos implementar estes conceitos no C#. Vimos também a diferença entre variáveis do tipo Reference Type e várias do tipo Value Type.

As variáveis reference type geralmente se referem a instâncias de classes. Conforme vimos, as classes são apenas especificações de uma determinada entidade do sistema que determinam o seu comportamento. Para criarmos uma instância de uma classe (um objeto) devemos criar uma variável do tipo desta classe e utilizar o comando new para alocar uma nova instância da classe. Podemos alocar quantas instâncias quantas forem necessárias, e para cada uma delas será alocado um espaço de memória para seus dados e seu funcionamento, conforme o exemplo a seguir.

```
public static void Main()
{
   Cliente cli1 = new Cliente();
   Cliente cli2 = new Cliente();
}
```

Introdução ao ciclo de vida

A grande diferença entre a criação de objetos no C# e a criação de objetos em Visual Basic 6.0 é que no C# não é necessário desalocar os objetos (definir o objeto para nothing). No momento em que criamos uma instância de qualquer objeto, o framework guarda uma referência desta instância. No momento em que as variáveis que referenciam esta instância saem de escopo e nenhuma variável aponta para aquela referência de objeto, o objeto é eliminado automaticamente.

O Garbage Collector

O processo responsável por eliminar referências de objetos não utilizadas é o garbage collector. Como o próprio nome diz, este recurso é responsável por desalocar qualquer recurso que não esteja sendo utilizado pelo programa.

O garbage colector é um processo não determinístico, ou seja, não é possível determinarmos quando o garbage collector irá executar para captar objetos não utilizados. Em geral o framework realiza este processo quando necessita alocar memória para novos objetos. Antes de alocar uma quantidade grande de memória, o garbage collector é executado para eliminar objetos não utilizados.

É possível interagir com o garbage collector através da classe System.GC. Esta classe permite inclusive a chamada de um método para forçar a execução do garbage collector.

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

É importante salientar que mesmo estando disponível, esta prática não é recomendada.

Finalização de objetos

Quando o garbage collector faz o processo de desalocação dos objetos, podemos fazer com

que uma função seja executada antes que o objeto seja eliminado. Essas funções são chamas funções destrutoras. Em geral utilizamos essas funções para destruirmos instâncias de objetos não gerenciáveis, tendo em vista que o framework não tem controle sobre este tipo de objeto.

A implementação de um destrutor é semelhante à sintaxe utilizada para construirmos um construtor, com a diferença que devemos utilizar o símbolo "~" na frente da declaração, conforme o exemplo a seguir.

```
public class Funcionario
{
   private Pxcs01vwClass objetoNaoGerenciavel;

   public Funcionario()
   {
      objetoNaoGerenciavel = new
         Pxcs01vwClass();
   }

   ~Funcionario()
   {
      objetoNaoGerenciavel.Deslogar();
   }
}
```

No exemplo acima temos uma classe que faz uso de um objeto COM (componente VB6). Este objeto aloca uma série de recursos não gerenciáveis que são utilizados durante a execução de métodos da classe. Para garantirmos que os recursos deste objeto COM sejam liberados devemos implementar um Pxcs01vwClass qualquer recurso alocado pela classe COM. Neste caso, chamamos um método deslogar para liberar estes recursos.

Disposable Classes

O único inconveniente da implementação utilizada acima é que o recurso do objeto COM só será liberado no momento em que o Garbage Collector for executado. Isto porque o método destrutor da classe só executa durante a execução do processo de garbage collection. Como não temos certeza de quando este processo irá executar, este recurso poderá ficar alocado por muito tempo, o que muitas vezes não é conveniente.

Para resolver este problema o .NET Framework dispõe de uma interface chamada IDisposable. Esta interface é um padrão do .NET Framework para a criação de funções de liberação de recursos. A função a ser implementada na interface chama-se Dispose.

Um usuário que tenha que utilizar a classe pode, ao invés de aguardar a execução do método de finalização, executar o método dispose para liberar os recursos não gerenciáveis da classe, conforme o exemplo a seguir.

```
public class Funcionario : IDisposable
{
   private Pxcs01vwClass objetoNaoGerenciavel;

   public Funcionario()
   {
      objetoNaoGerenciavel = new
```

```
            Pxcs01vwClass();
        }

        ~Funcionario()
        {
            if(objetoNaoGerenciavel != null)
                objetoNaoGerenciavel.Deslogar();
        }

        public void Dispose()
        {
            if (objetoNaoGerenciavel != null)
            {
                objetoNaoGerenciavel.Deslogar();
                objetoNaoGerenciavel = null;
            }
        }
    }
```

Note que implementamos o método Dispose de forma que ele possa ser executado mais de uma fez. Fazemos isto definindo a variável do objeto não gerenciável para null e testando nas próximas execuções. É importante salientar que NÃO É NECESSÁRIO definir a variável para null para desalocar um objeto, este artifício foi utilizado apenas por uma questão de lógica.

[Exercício 4.1 da lista de exercícios]

6. Tratamento estruturado de exceções

A importância do tratamento estruturado de exceções

Nos capítulos anteriores verificamos alguns conceitos sobre a orientação a objetos e como podemos implementar estes conceitos no C#. Vimos também a diferença entre variáveis do tipo Reference Type e várias do tipo Value Type.

Neste capítulo vamos ver como funciona a estrutura de tratamento de erros no C#. O tratamento de erros no C# funciona de uma forma um pouco diferente em relação ao Visual Basic 6.0. No Visual Basic utilizamos o comando On Error com um apontador Goto para uma região onde o erro deveria ser tratado. No C# trabalhamos com o conceito de tratamento estruturado de exceções, onde devemos envolver o código onde existe a possibilidade de ocorrer uma exceção, além de informar os tipos de exceção que devem ser tratadas.

O tratamento de exceções é fundamental em qualquer aplicação, tendo em vista que qualquer aplicação está sujeita a erros, sejam estes de programação ou de alguma falha de ambiente.

Exemplo simples de tratamento estruturado de exceções

Para entendermos melhor o tratamento de exceções, vamos avaliar o código abaixo:

```
StreamReader sr = new StreamReader("C:\\TESTE.TXT");

string linha = "";

while (sr.Peek() != -1)
{
   linha = sr.ReadLine();
   string[] numeros = linha.Split(',');
   int soma = int.Parse(numeros[0]) + int.Parse(numeros[1]);
   Console.WriteLine(soma.ToString());
}
sr.Close();
```

No exemplo acima temos um trecho de código onde fazemos a leitura de um arquivo e fazemos a soma de alguns elementos separados por vírgula dentro do arquivo.

Nesse trecho de código pode ocorrer uma série de problemas como, por exemplo, falta de permissão no arquivo, erro na conversão dos números, etc. Para criarmos um bloco de código com tratamento estruturado de exceções, devemos envolver o bloco acima em um comando try. O comando try indica que o código deverá ser testado para verificação de exceções, ou seja, caso uma exceção ocorra, o código é desviado para um bloco de código de tratamento de exceções onde todas as informações sobre o erro podem ser obtidas. O exemplo a seguir mostra o mesmo trecho de código utilizado tratamento estruturado de exceções.

```
try
{
   StreamReader sr = new StreamReader("C:\\TESTE.TXT");

   string linha = "";

   while (sr.Peek() != -1)
   {
      linha = sr.ReadLine();
      string[] numeros = linha.Split(',');
      int soma = int.Parse(numeros[0]) + int.Parse(numeros[1]);
```

```
            Console.WriteLine(soma.ToString());
        }
        sr.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
```

System-Level Exceptions

No exemplo que utilizamos anteriormente você deve ter percebido que o código com tratamento de erro foi envolvido em um bloco pelo comando try. Após o bloco do comando try, podemos observar um comando catch. No comando catch devemos informar o tipo de exceção que queremos tratar. Caso esta exceção ocorra, será tratada pelo bloco de código indicado.

```
try
{
...
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

No exemplo acima, estamos tratando uma exceção do tipo Exception. A exceção do tipo exception é a exceção mais genérica possível disponível no C#. Com esta exceção podemos tratar qualquer tipo de erro ocorrido dentro do sistema. A desvantagem deste tipo de exceção é que por ser genérica, muitas vezes ela pode não trazer detalhes importantes sobre o erro ocorrido.

O tipo exception na realidade representa uma classe definida no .NET Framework. Toda e qualquer exceção que ocorre deve ser derivada do tipo Exception. Dentro de um sistema .NET podemos ter dois tipos de exceções, as System Exceptions e as ApplicationExceptions. As system exceptions são intrínsecas ao .NET Framework e dela derivam uma série de outras exceções especializadas.

Podemos utilizar mais de um bloco catch dentro de um bloco try, a fim de tratar mais de um tipo de exceção. Poderíamos por exemplo tratar exceções relacionadas a problemas no filesystem, a problemas de permissão ou mesmo problemas de conversão de dados. Para capturarmos estas diferentes exceções, devemos utilizar blocos catch específicos para este tipo de exceção, conforme o exemplo a seguir.

```
try
{
    StreamReader sr = new StreamReader("C:\\TESTE.TXT");

    string linha = "";

    while (sr.Peek() != -1)
    {
        linha = sr.ReadLine();
        string[] numeros = linha.Split(',');
        int soma = int.Parse(numeros[0]) + int.Parse(numeros[1]);
        Console.WriteLine(soma.ToString());
    }
    sr.Close();
}
catch (FileNotFoundException ex)
```

```
{

    Console.WriteLine("Arquivo não encontrado." + ex.FileName);
}
catch (FormatException ex)
{
    Console.WriteLine("Erro convertendo dados");
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

No exemplo acima temos o tratamento para diversos tipos de exceção. Se o arquivo não existir no disco, a exceção gerada será tratada pelo bloco com a FileNotFoundException. Se obtivermos um erro de conversão de dados, o erro será tratado pelo bloco com a FormatException. Caso ocorra qualquer outro tipo de erro, este será tratado pelo bloco com a Exception.

O C# sempre tenta identificar as exceções na ordem em que elas foram programadas no código, ou seja, devemos sempre informar as exceções mais específicas para depois informarmos as exceções mais genéricas. Se por exemplo colocássemos o bloco com a classe Exception antes de qualquer outro bloco catch, todas as exceções cairiam neste bloco, pois ele foi colocado antes dos blocos de tratamento para exceções específicas.

É importante notar também que em cada bloco catch temos disponível uma variável ex que contem as informações sobre a exceção gerada. Podemos ter propriedades diferentes para cada tipo de exceção. No caso de uma FileNotFoundException, por exemplo, temos uma propriedade chamada FileName que indica o arquivo que não foi encontrado.

Application Exception

As ApplicationExceptions são exceções que podemos ser criadas dentro de nossa própria aplicação. Assim como o .NET Framework possui exceções específicas, podemos criar classes que representam exceções que queremos gerar dentro de nosso sistema. Para criarmos uma exceção customizada, devemos criar uma nova classe que herde as características da classe ApplicationException. O exemplo a seguir mostra uma exceção customizada.

```
public class PxuException : ApplicationException
{
    public string retCOM;
    public string retUSU;
    public string codFalha;

    public PxuException(string rc, string ru, string cf)
    {
      retCOM = rc;
      retUSU = ru;
      codFalha = cf;
    }
}
```

Após criarmos as exceções, podemos lançar estas exceções em nosso código através do comando throw. Este comando faz com que o sistema lance a exceção que deve ser tratada pela rotina chamadora. Caso não seja tratada, a exceção faz com que o sistema seja abortado. A seguir

podemos ver um exemplo do comando throw.

```
throw new PxuException("00", "01", "1235");
```

O bloco finally

Além do bloco try e dos blocos batch, o tratamento estruturado de exceções ainda possui um outro bloco opcional que pode ser utilizado, o bloco finally. O bloco finally serve para colocarmos algum bloco de código que deve ser executado independente de o processo ter executado com êxito ou de ter ocorrido uma exceção. Este tipo de bloco é útil para as situações em que devemos fechar conexões, fechar arquivos ou mesmo liberar algum recurso caro para o sistema operacional.

A seguir podemos ver a implementação de um bloco finally, onde fazemos o fechamento de um arquivo caso ele tenha sido criado e esteja aberto.

```
StreamReader sr = null;
try
{
   sr = new StreamReader("C:\\TESTE.TXT");

   string linha = "";

   while (sr.Peek() != -1)
   {
      linha = sr.ReadLine();
      string[] numeros = linha.Split(',');
      int soma = int.Parse(numeros[0]) + int.Parse(numeros[1]);
      Console.WriteLine(soma.ToString());
   }
   sr.Close();
}
catch (FileNotFoundException ex)
{

   Console.WriteLine("Arquivo não encontrado." + ex.FileName);
}
catch (FormatException ex)
{
   Console.WriteLine("Erro convertendo dados");
}
catch (Exception ex)
{
   Console.WriteLine(ex.ToString());
}
finally
{
   if (sr != null)
      sr.Close();
}
}
```

[Exercício 5.1 da lista de exercícios]

7. Delegates e eventos

Introdução

A maioria de nós já foi exposta a algum tipo de aplicação baseada em eventos. O C# permite o uso deste tipo de funcionalidade através de dois recursos, os eventos e os delegates. Neste capítulo veremos como funciona o mecanismo de criação de delegates e o mecanismo de subscrição de eventos.

Delegates

Delegates são similares aos ponteiros de função disponíveis nas linguagens C e C++. Utilizando um delegate podemos encapsular uma referencia a um método dentro de um objeto. O objeto de delegate pode ser utilizado em qualquer ponto do código para chamar o método referenciado, sem necessariamente saber em tempo de compilação qual o método será chamado.

Chamando uma função sem delegates

Na maioria dos casos, quando queremos chamar uma função, fazemos esta chamada de forma direta. No exemplo a seguir temos uma classe com uma função chamada Process. Na classe Teste, temos a chamada a esta função feita diretamente através da instancia da classe.

```
public class MinhaClasse
{
   public void Process()
   {
      Console.WriteLine("Process() inicio");
      Console.WriteLine("Process() fim");
   }
}

public class Teste
{
   static void Main(string[] args)
   {
      MinhaClasse a = new MinhaClasse();
      a.Process();
   }
}
```

O exemplo acima funciona para a maioria das situações. Algumas vezes, porém, não queremos chamar uma função de forma direta, precisamos passar uma referencia desta função para que outra classe faça esta chamada. Isto é extremamente útil em sistemas baseados em eventos, onde temos que definir uma função em um formulário, por exemplo, para tratar a resposta a um click de um botão na tela.

Um delegate simples

Um ponto interessante sobre os delegates é que não existe qualquer restrição ao tipo de objeto ao qual ela está referenciando, o que importa é que a função referenciada possua os mesmos parâmetros e o mesmo tipo de retorno. A implementação de delegates também é conhecida por métodos anônimos.

A sintaxe para criarmos um delegate é semelhante ao código a seguir.

```
delegate result-type identifier ([parameters]);
```

A seguir temos um exemplo de implementação com delegates.

```
public delegate void SimpleDelegate ()
```

O comando acima declara um delegate com o nome de SimpleDelegate, que não recebe qualquer parâmetro e não retorna nenhum valor.

```
public delegate int ButtonClickHandler (object obj1, object obj2)
```

O comando acima representa um delegate que recebe dois parâmetros do tipo object e retorna um tipo inteiro.

Os delegates irão permitir que especifiquemos "o que" a função que vamos chamar deve aceitar, e não "qual" função iremos chamar.

Associando uma função estática a um delegate

No exemplo a seguir temos a implementação de um código simples utilizando delegates.

```
public delegate void DelegateExemplo();

class TestDelegate
{
   public static void MinhaFuncao()
   {
      Console.WriteLine("I was called by delegate ...");
   }

   public static void Main()
   {
      DelegateExemplo DelegateExemplo = new DelegateExemplo(MinhaFuncao);

      DelegateExemplo();
   }
}
```

No exemplo acima, note que construimos o delegate passando como parâmetro a função que será chamada. Para chamarmos a função apontada para o delegate, basta utilizar o nome da variável que contém o delegate e os parenteses, exatamente como uma função normal.

```
public class MinhaClasse
{

   public delegate void LogHandler(string message);

   public void Process(LogHandler logHandler)
   {
      if (logHandler != null)
      {
         logHandler("Process() begin");
      }

      if (logHandler != null)
      {
         logHandler("Process() end");
      }
   }
}
```

```
public class TestApplication
{
   static void Logger(string s)
   {
      Console.WriteLine(s);
   }

   static void Main(string[] args)
   {
      MinhaClasse MinhaClasse = new MinhaClasse();

      MinhaClasse.LogHandler myLogger = new MinhaClasse.LogHandler(Logger);
      MinhaClasse.Process(myLogger);
   }
}
```

No exemplo acima temos uma classe chamada MinhaClasse que possui uma função chamada Process. A função aceita um parâmetro do tipo delegate, que representa uma função que será chamada para notificar sobre o andamento do processo. Na outra classe, onde a aplicação é executada, foi criada uma função estática com o nome de logger. Note que ao chamarmos a função Process passando uma instancia de um delegate que aponta para a função estática Logger da classe de execução do programa. Esta função simplesmente mostra os valores na tela de console.

Associando uma função de uma classe a um delegate

Agora vamos alterar o código acima para utilizarmos o delegate com uma função pertencente a uma instancia de classe. Observe que nesta situação não alteramos em nada o código da classe MinhaClasse, ela continua fazendo a chamada a uma função de log. A grande diferença desta implementação é que a função de log chamada aponta para uma função da classe FileLogger, que grava a log em um arquivo ao invés de imprimir na tela.

```
public class MinhaClasse
{
   public delegate void LogHandler(string message);

   public void Process(LogHandler logHandler)
   {
      if (logHandler != null)
      {
         logHandler("Process() inicio");
      }
      if (logHandler != null)
      {
         logHandler("Process() fim");
      }
   }
}

public class FileLogger
{
   FileStream fileStream;
   StreamWriter streamWriter;

   // Construtor
   public FileLogger(string filename)
   {
      fileStream = new FileStream(filename, FileMode.Create);
      streamWriter = new StreamWriter(fileStream);
   }
```

```
    public void Logger(string s)
    {
        streamWriter.WriteLine(s);
    }

    public void Close()
    {
        streamWriter.Close();
        fileStream.Close();
    }
}

public class TestApplication
{
    static void Main(string[] args)
    {
        FileLogger fl = new FileLogger("process.log");
        MinhaClasse MinhaClasse = new MinhaClasse();

        MinhaClasse.LogHandler myLogger = new MinhaClasse.LogHandler(fl.Logger);
        MinhaClasse.Process(myLogger);
        fl.Close();
    }
}
```

Apontando para mais de uma função

Outro aspecto interessante sobre os delegates é que eles podem apontar não só para uma mas para mais de uma função. Se quiséssemos que a função gravasse a log no arquivo e, além disso, exibisse as informações na tela de console, poderíamos inserir novamente a função estática no código anterior e alterar a criação do delegate para que ela ficasse da seguinte forma.

```
    static void Main(string[] args)
    {
        FileLogger fl = new FileLogger("process.log");

        MinhaClasse MinhaClasse = new MinhaClasse();

        MinhaClasse.LogHandler myLogger = new MinhaClasse.LogHandler(fl.Logger);
        myLogger += new MinhaClasse.LogHandler(Logger);
        MinhaClasse.Process(myLogger);
        fl.Close();
    }
```

No exemplo acima, utilizamos o operador += para adicionar uma função ao delegate. Quando o delegate for chamado pela classe MinhaClasse, irá chamar as duas funções associadas.

8. Entendendo Generics

Introdução

Com o lançamento do .NET Framework 2.0, o C# ganhou suporte a uma nova funcionalidade, chamada de Generics. Os generics permitem ao desenvolvedor a criação de "placeholders" (também conhecidos como parâmetros de tipos) para os tipos utilizados em parâmetros de métodos ou mesmo para valores de retorno de funções. Estes tipos são definidos na criação do objeto em tempo de execução, ao invés de em tempo de compilação.

Revisando o conceito de boxing e unboxing

Para entender os benefícios do uso dos generics, é importante revisarmos o conceito de boxing e unboxing. O conceito de boxing e unboxing ocorre quando temos uma variável que está declarada no stack passada para uma alocação do tipo heap. Este processo ocorre quando convertemos uma variável do tipo Value Type para um tipo Reference type, no caso para o tipo System.Object.

Nestas situações, quando queremos passar um valor para um método ou classe que aceita como parâmetro um tipo object, o run-time do .net precisar fazer uma operação de boxing (conversão para um "Box") para guardar este valor e tratá-lo como um reference type.

Um caso em que isto ocorre com freqüência, é no uso da classe ArrayList. A classe arraylist permite a inserção de itens do tipo object, o que obriga o run-time a fazer uma operação de boxing sempre que inserimos um novo item na lista. O mesmo acontece quando removemos o item da lista, onde se faz necessária uma operação de unboxing.

Estas operações são extremamente custosas para a aplicação em termos de desempenho, pois a conversão ocorre toda a vez que acessamos um objeto da lista, conforme o exemplo a seguir.

```
ArrayList lista = new ArrayList();

// boxing do número 10
// em um object
lista.Add(10);

// unboxing da referência
// object para o valor inteiro
int valor = (int) lista[0];
```

O namespace System.Collections.Generics

A biblioteca System.Collections.Generics possui itens semelhantes a sua irmã "não genérica" System.Collections. A diferença entre as duas bibliotecas é que a biblioteca genérica permite a criação de coleções tipas, evitando as operações de boxing e unboxing e aumentando o desempenho do sistema.

Dentre as classes e interfaces disponíveis na biblioteca System.Collections.Generics, podemos enumerar as seguintes.

• ICollection<T>

• IComparer<T>

• IDictionary<K, V>

• IEnumerable<T>

• IEnumerator<T>

• IList<T>

Analisando a classe List<T>

Analogamente a classe ArrayList, a classe List da biblioteca genérica permite a criação de um coleção para armazenamento de um conjunto de dados quaisquer. A diferença entre as duas bibliotecas é que a biblioteca genérica nos permite informar o tipo de dados o que queremos armazenar na coleção. Esta informação é indicada na declaração da coleção pelo parâmetro T, onde devemos informar o tipo de dados a ser utilizado.

O exemplo a seguir cria uma lista de números inteiros.

```
List<int> lista = new List<int>();

// não ocorre boxing, pois
// definimos que a lista é de inteiros
lista.Add(10);

// não é necessário utilizar
// funções de conversão
int valor = lista[0];
```

Note que em tempo de execução, definimos que a lista utilizaria valores inteiros, informando o tipo int no parâmetro T da coleção. Os parâmetros para tipos genéricos devem ser informados entre os símbolos de menor que e maior que (< >).

Para entendermos como a linguagem monta este tipo de estrutura genérica, vamos analisar a assinatura dos métodos da própria classe List<T>.

```
public class List<T> :
IList<T>, ICollection<T>, IEnumerable<T>,
IList, ICollection, IEnumerable
{
...
public void Add(T item);
public IList<T> AsReadOnly();
public int BinarySearch(T item);
public bool Contains(T item);
public void CopyTo(T[] array);
public int FindIndex(System.Predicate<T> match);
public T FindLast(System.Predicate<T> match);
public bool Remove(T item);
public int RemoveAll(System.Predicate<T> match);
public T[] ToArray();
public bool TrueForAll(System.Predicate<T> match);
public T this[int index] { get; set; }
}
```

Note que em todos os pontos onde antes indicávamos object ou mesmo um tipo específico, agora especificamos a letra T. Na declaração da classe, informamos também o parâmetro <T> após a declaração da classe, o que indica que a classe é genérica, e utiliza um tipo que será definido na utilização do objeto.

```
public static void GravaLog<T>(T informacao)
```

```
{
    string mensagem = informacao.ToString();
    // ... grava log
}
```

No exemplo acima, indicamos que o método GravaLog utiliza um tipo genérico. O tipo genérico é utilizado como parâmetro para passarmos a informação que será gravada na log. No caso de métodos genéricos, devemos informar o tipo de dado na chamada do método, conforme o exemplo a seguir.

```
public void Main()
{
    GravaLog<string>("TESTE");
    GravaLog<int>(10);
    GravaLog<DateTime>(DateTime.Now);
}
```

[Exercício 7.1 da lista de exercícios]

9. .NET Assemblys

Introdução

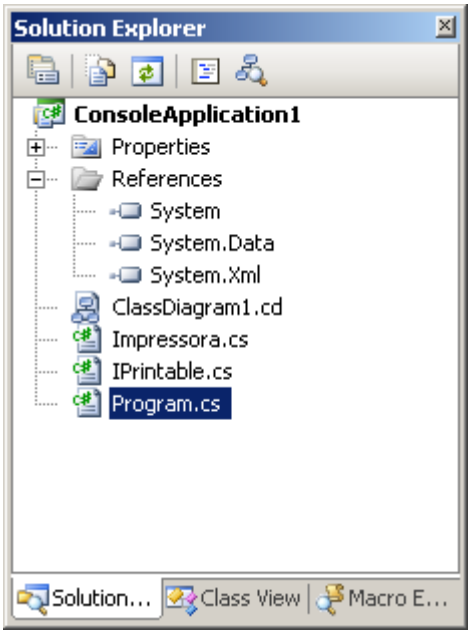
As aplicações que criamos até o momento geravam apenas um módulo, que quando executado produzia um resultado qualquer. Na prática, quando estamos trabalhando com um projeto .NET, a tendência é criarmos uma solução de forma mais modularizada, não colocando os componentes em um único projeto, mas sim em vários projetos menores que facilitam a manutenção e o reaproveitamento do código.

Para criarmos estes projetos de forma distribuída, devemos fazer uso dos assemblys. Os assemblys são nada mais nada menos que um conjunto de arquivos de código compilados que contém uma identidade, uma versão e uma assinatura. Podemos entender por assembly um arquivo compilado em .NET, seja ele um .EXE ou um arquivo .DLL.

A reutilização de código e os assemblys

Você pode ter pensado que toda a lógica que utilizamos em nossas aplicações até o momento estão compiladas dentro o arquivo gerado como resultado da compilação da aplicação. Na prática, qualquer aplicação .NET sempre faz referência a outros assemblys.

Se observarmos o Solution Explorer, podemos ver que existe uma pasta chamada References, que quando expandida mostra itens. Estes itens representam os assemblys que são utilizados pelo programa.



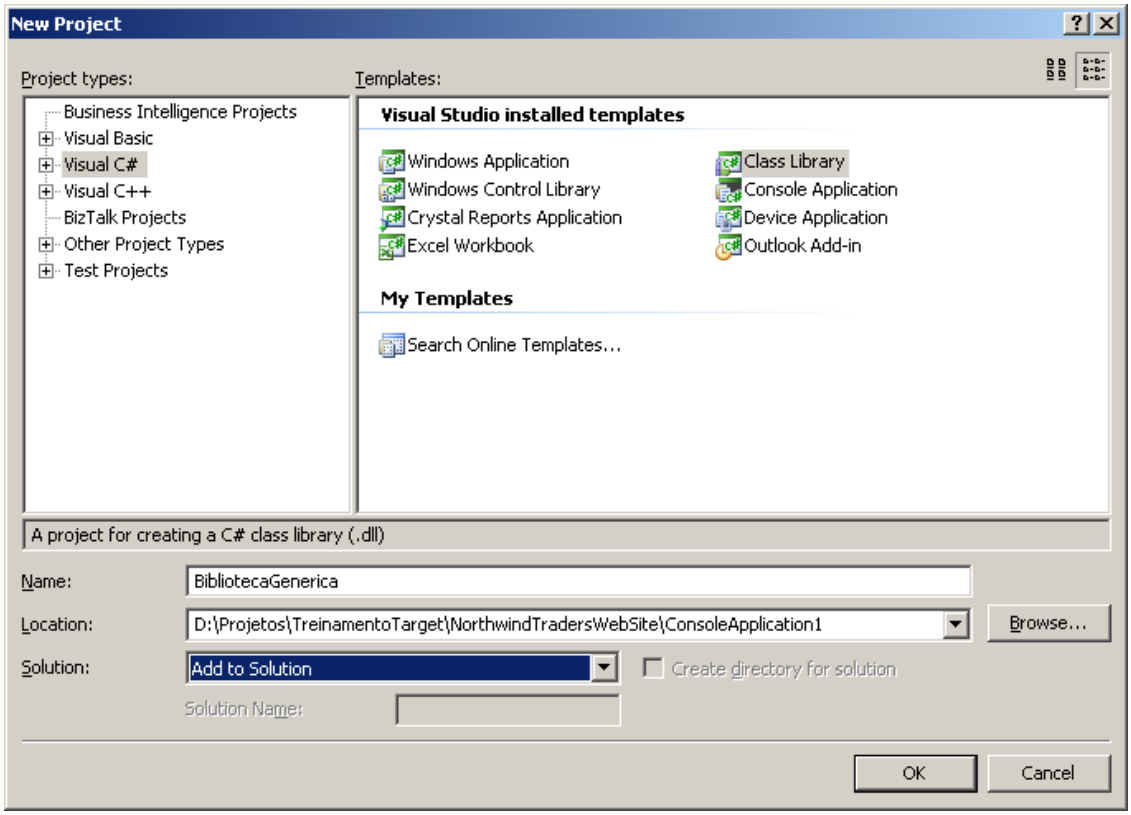
Criando um assembly

Os assemblys são representados por projetos do tipo Class Library. Estes projetos permitem a criação de um conjunto de classes em um módulo à parte, que terá como resultado final um arquivo com a extensão.dll (um assembly). Este assembly poderá ser posteriormente utilizado por outras aplicações.

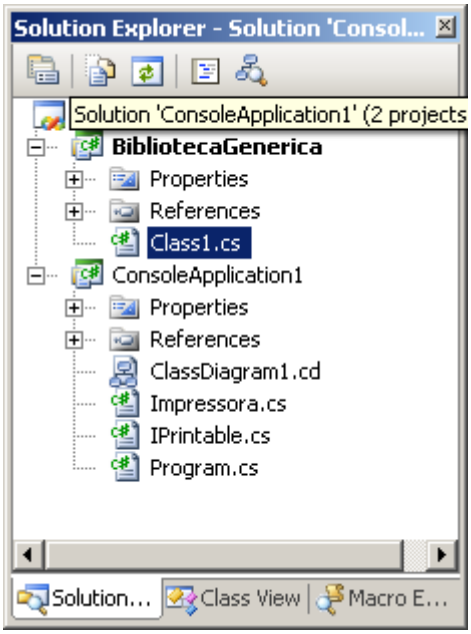
Para criarmos um assembly, devemos criar um novo projeto no Visual Studio, o que pode ser

feito seguindo os seguintes passos.

1. Acesse o menu File do Visual Studio
2. Selecione a opção New Project
3. Selecione o tipo de projeto Class Library

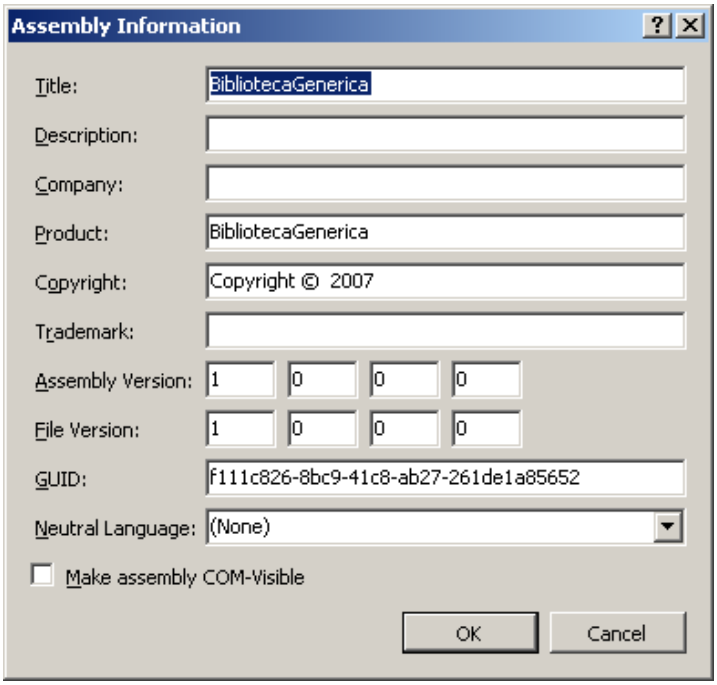


4. Informe o nome da biblioteca e clique em ok. Caso você já tenha um projeto aberto, poderá fazer com que este componente seja adicionado a mesma solução, desta forma você poderá trabalhar com vários projetos de diferentes componentes ao mesmo tempo.



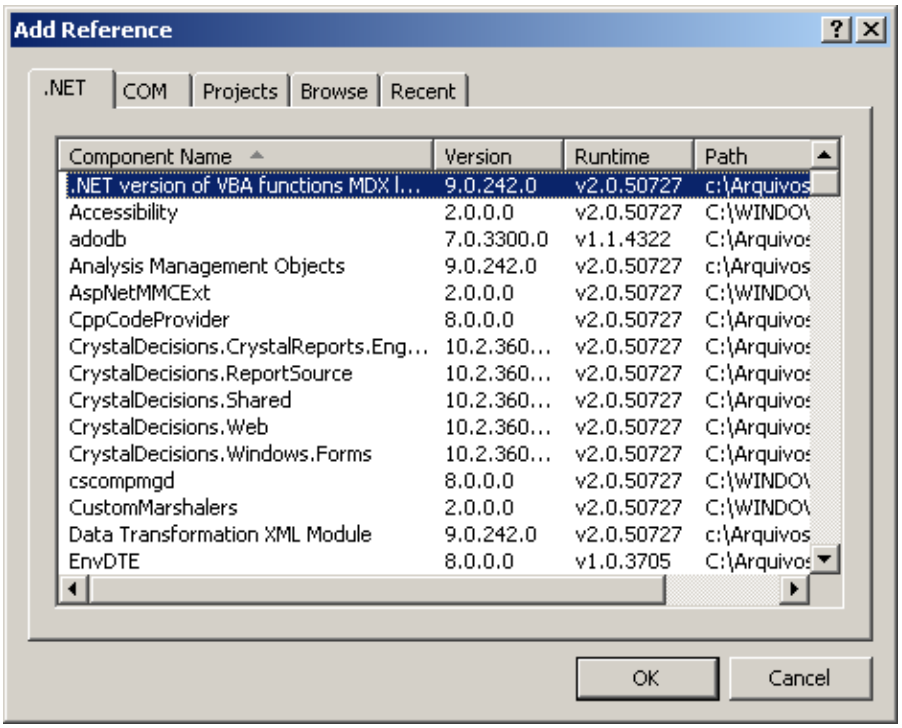
Alterando as propriedades dos assemblys

No solution explorer, podemos observar a existência de um item chamado Properties. Neste item podemos informar alguns itens adicionais para o assembly como nome do fabricante, versão, linguagem, etc. Estas informações podem ser acessadas através do botão "Assembly Information".



Referenciando um assembly

Após criarmos o assembly, podemos referenciá-lo em outros projetos a fim de utilizar suas classes. Para isso, basta acessarmos o solution explorer do projeto em questão e clicarmos com o botão direito sobre a pasta References, selecionando a opção Add Reference.



As referencias podem ser adicionadas a partir de uma DLL em uma pasta qualquer do sistema operacional através da opção browse, podem ser adicionadas a partir de um projeto da mesma solução, através da aba “Projects” ou podem ser adicionadas do repositório global de assemblys (o GAC) através da aba .NET.

Private Assemblys e Shared Assemblys

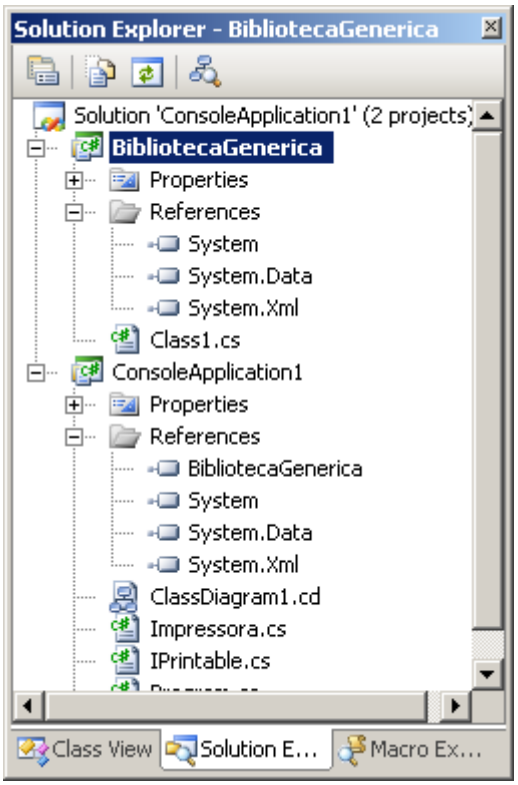
Durante o desenvolvimento de um projeto, note que quando você adiciona a referência de um assembly ao projeto e compila o projeto, o Visual Studio automaticamente copia a DLL resultante da compilação para a pasta de compilação do projeto (em geral a pasta BIN).

A pasta do projeto é o repositório padrão para componentes de programa. Quando o .NET Framework carrega o programa para execução ele automaticamente carrega todas as DLLs presentes na pasta, considerando que estas DLLs são utilizadas pela aplicação para suas operações.

As dlls que se encontram na pasta da aplicação são chamadas private assemblys, pois são dlls utilizadas apenas para aquela aplicação, ou seja, para que a aplicação faz uso delas é preciso que elas estejam na pasta do projeto da aplicação.

Analisando as referências

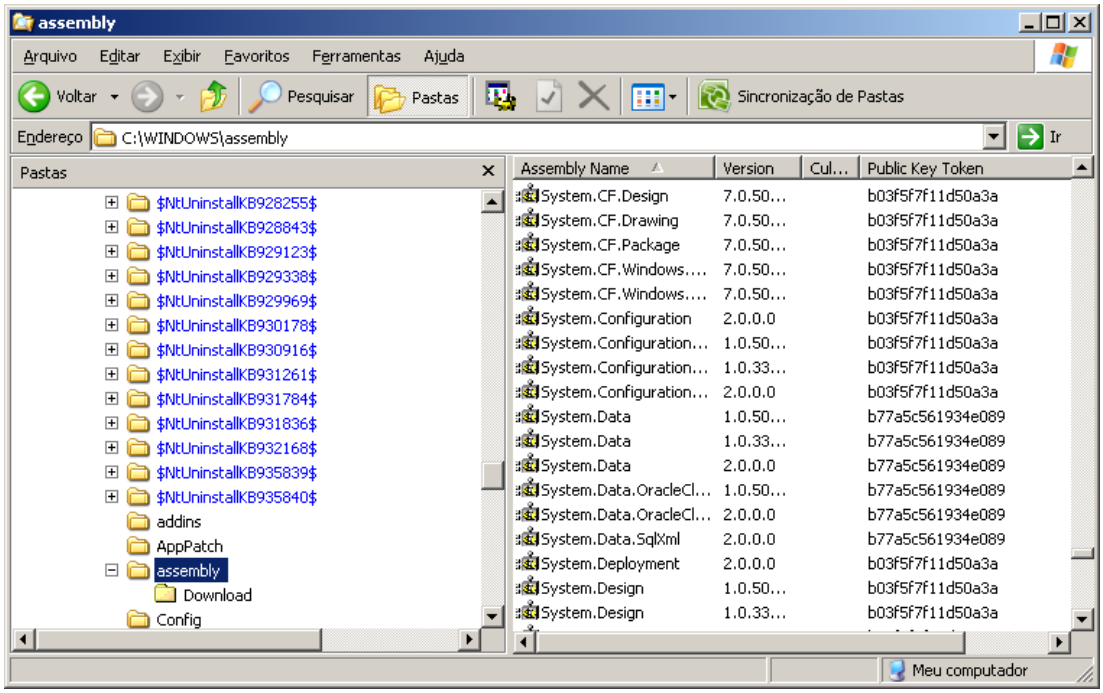
Em um project, temos como verificar quais assemblys nosso projeto está fazendo uso. Para isto, basta clicar expandir a pasta de References do Solution Explorer.



Se tentarmos incluir um assembly da aba “.NET” ao invés da aba Project ou Browse, iremos perceber um comportamento diferente para estes componente. Se adicionarmos ao projeto o componente System.Transactions por exemplo, ao compilarmos o projeto, podemos perceber que a DLL deste componente não é adicionada a pasta de compilação do projeto. Isto ocorre porque esses assemblys são chamados assemblys compartilhados (shared assemblys). Tais assemblys estão

disponíveis para toda a máquina na qual estão registrados. Um exemplo de shared assemblys são os próprios componentes do .NET Framework (que iniciam com prefixo System). Estes componentes não requerem uma cópia na pasta do projeto, pois estão disponíveis globalmente para a máquina.

Para verificarmos o conteúdo do Global Assembly Cache, basta abrirmos o Windows Explorer e acessarmos a pasta \WINDOWS\Assembly.



[Exercício 8.1 da lista de exercícios]

10. Acesso ao banco de dados

Encapsulamento do acesso ao banco de dados

No curso de C# temos disponível uma biblioteca de classes que auxiliam o desenvolvimento de classes para acesso ao banco de dados, a Pxcoiexn.dll disponível em c:\soft\pxc\bin ou na pasta temporária do treinamento.

Classes auxiliares

Para entender como acessar, primeiro vamos entender como funcionam as classes e estruturas da biblioteca Bergs.Pxc.Pxcoiexn.

```
// Summary:
//     Estrutura de retorno com controle de conversão
//
// Type parameters:
//   T:
public struct Retorno<T>
{
    // Summary:
    //     Dados retornados pelo método chamado
    public readonly T Dados;
    //
    // Summary:
    //     Mensagem retornada pelo método
    public readonly Mensagem Mensagem;
    //
    // Summary:
    //     Indica se o método foi realizado com sucesso
    public readonly bool Ok;
}
```

A estrutura acima contempla várias informações agrupadas que seguem: T é o tipo de retorno que desejamos informar, Dados é o dado de retorno, Ok indica se o método retornou com sucesso ou com erro, e Mensagem é a mensagem de retorno.

```
// Summary:
//     Classe de retorno de mensagens
public class Mensagem
{
    // Summary:
    //     Conteúdo da mensagem
    protected string mensagem;

    // Summary:
    //     Construtor da classe
    public Mensagem();
    //
    // Summary:
    //     Construtor da classe
    //
    // Parameters:
    //   excecao:
    public Mensagem(Exception excecao);

    // Summary:
    //     Mensagem retornada para o operador
    public string ParaOperador { get; }

    // Summary:
    //     Mensagem
    public override string ToString();
}
```

A classe de mensagens serve para retornar uma mensagem ao operador. Novas classes de mensagens podem ser criadas herdando dessa classe.

Para conseguir retornar o tipo Retorno<T>, devemos criar uma classe que herde da classe Aplicacao. Abaixo segue um exemplo da forma de utilização e a comparação entre 2 métodos que realizam a mesma função.

```
using Bergs.Pxc.Pxcoiexn;
//...
   class Teste : Aplicacao
   {
      public Int32 ContarEspacosInt32(String texto)
      {
         return texto.Split(' ').Length - 1;
      }

      public Retorno<Int32> ContarEspacosRetornoInt32(String texto)
      {
         try
         {
            Int32 retorno = texto.Split(' ').Length - 1;
            return this.RetornarSucesso<Int32>(retorno, new OperacaoRealizadaMensagem());
         }
         catch (Exception e)
         {
            return this.RetornarFalha<Int32>(new Mensagem(e));
         }
      }
   }
```

Como podemos verificar, o método com Retorno<Int32> é mais robusto, onde tem o encapsulamento do tratamento de exceção. Abaixo segue um exemplo da chamada desse método.

```
      static void Exemplo()
      {
         Teste teste = new Teste();
         String texto = "Texto com espaços para teste de Retorno<Int32>";
         Retorno<Int32> retornoContar = teste.ContarEspacosRetornoInt32(texto);
         if (retornoContar.Ok)
            Console.WriteLine("Quantidade de espaços = {0}", retornoContar.Dados);
         else
            Console.WriteLine(retornoContar.Mensagem);
      }
```

Digamos que fosse necessário dar uma mensagem diferenciada caso a quantidade de espaços fosse igual à zero, como por exemplo “Nenhum espaço encontrado”. A classe Mensagem não dá recursos para isso, somente se a mesma fosse herdada. Uma possibilidade é como está feito no código abaixo.

```
   class MensagemExemplo : Mensagem
   {
      public MensagemExemplo(String mensagem)
      {
         this.mensagem = mensagem;
      }
   }
```

Então com essa classe, podemos modificar o método ContarEspacosRetornoInt32 conforme feito abaixo.

```
      public Retorno<Int32> ContarEspacosRetornoInt32(String texto)
      {
         try
         {
```

```
            Int32 retorno = texto.Split(' ').Length - 1;
            if (retorno == 0)
                return this.RetornarFalha<Int32>(new MensagemExemplo("Nenhum espaço encontrado"));
            return this.RetornarSucesso<Int32>(retorno, new OperacaoRealizadaMensagem());
        }
        catch (Exception e)
        {
            return this.RetornarFalha<Int32>(new Mensagem(e));
        }
    }
```

Acesso ao banco de dados

Para acessar uma tabela utilizamos uma classe que herda de "Tabela" para mapeamento dos campos da tabela. Cada campo da tabela é representado pela classe CampoTabela<T>.

Exemplo:

```
public class TOTabela1 : Tabela
{
    private CampoTabela<Int32> codigo;
    //...
    public override void PopularRetorno(Linha linha)
    {
        foreach (Campo campo in linha.Campos)
        {
            switch (campo.Nome)
            {
                case "CODIGO":
                    this.codigo = this.LeCampoTabela<Int32>(campo.Conteudo);
                    break;
    //...
                default:
                    break;
            }
        }
    }
}
```

O campo "codigo" representa o campo "CODIGO" da tabela "TABELA1". O método PopularRetorno é acionado ao realizar uma consulta na tabela, onde recebemos uma coleção de campos representado por uma linha da tabela.

Para atribuir um valor válido ao campo "Codigo", deve-se fazer como abaixo.

```
TOTabela1 toTabela1 = new TOTabela1();
//...
toTabela1.Codigo = 1;
```

Para atribuir null a um campo da tabela deve-se fazer como exemplificado abaixo.

```
toTabela1.Codigo = new CampoTabela<Int32>(null);
```

Vamos entender a classe CampoTabela<T>.

```
// Summary:
//     Estrutura que representa um campo da tabela
//
// Type parameters:
//   T:
//     Tipo de dado do campo
public class CampoTabela<T> : ICampo
{
    // Summary:
    //     Retorna o conteúdo do campo, caso não tenha sido setado, retorna null
    public object Conteudo { get; set; }
```

```
    //
    // Summary:
    //     Retorna informação de que o campo foi setado, caso tenha sido atribuído
    public bool FoiSetado { get; }
    //
    // Summary:
    //     Retorna true se possuir conteúdo diferente de null (caso não tenha sido setado,
    //     retorna false)
    public bool TemConteudo { get; }
    // Summary:
    //     Retorna o conteúdo do campo, caso não tenha sido setado, retorna default(T)
    public T LerConteudoOuPadrao();
    //...
}
```

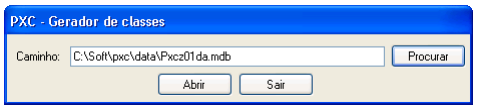
O campo FoiSetado indica se houve atribuição ao campo, caso ele tenha sido inicializado. Já o campo TemConteudo indica se o campo tem conteúdo (se FoiSetado é verdadeiro) e se o conteúdo é diferente de null.

O campo Conteudo retorna o valor contido no campo, que pode ser null. Já o método LerConteudoOuPadrao retorna o valor atribuído ao conteúdo, caso seja null, então retornará o valor default do tipo T.

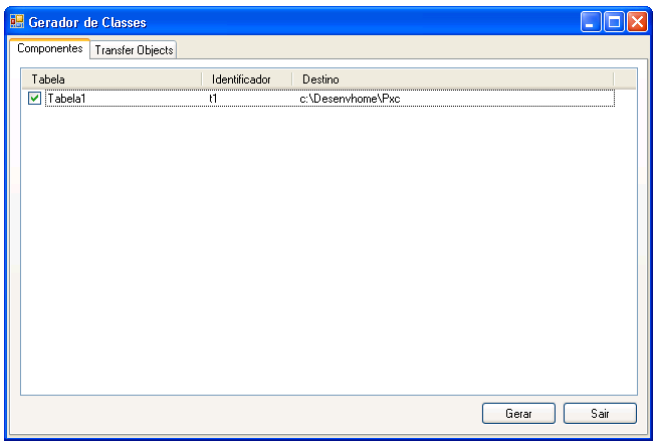
Para executar um comando ao banco de dados, devemos criar uma classe que herde de "BD".

Para entender melhor essa classe, utilize o programa Pxcpgpxn.exe para gerar as classes de acesso ao banco de dados e as classes de mapeamento das tabelas.

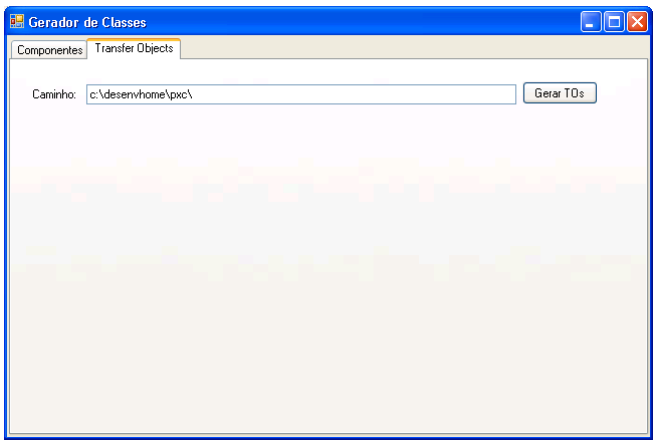
Primeiro informe o caminho do Access.



Depois dê um duplo clique sobre a tabela que deseja gerar, informe o "id" da tabela e o caminho para geração da classe, selecione e clique em gerar.



Para gerar a classe de mapeamento com a tabela, clique na aba Transfer Objects, informe o caminho para geração das classes e clique em "Gerar TOs".

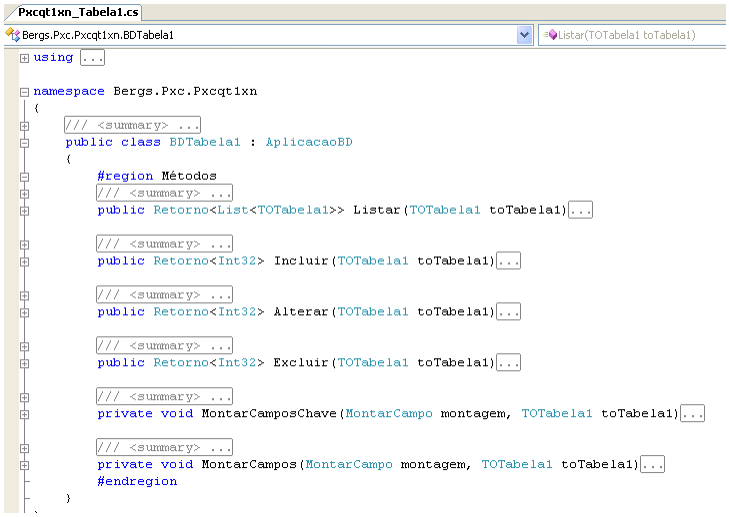


A classe TOTabela1 é gerada conforme visto anteriormente para cada um dos campos.

A classe BDTabela1 é gerada herdando da classe AplicacaoBD encapsulando o acesso ao banco de dados.

```
public class BDTabela1 : AplicacaoBD
{
    #region Métodos
   /// <summary>
   /// Executa o comando de consulta na tabela
   /// </summary>
   /// <param name="toTabela1">Campos para pesquisa na tabela</param>
   /// <returns>Retorna a lista consultada</returns>
   public Retorno<List<TOTabela1>> Listar(TOTabela1 toTabela1)
   {
      try
      {
   //...
         return this.RetornarSucesso<List<TOTabela1>>(lista, new OperacaoRealizadaMensagem());
      }
      catch (Exception e)
      {
         return this.RetornarFalha<List<TOTabela1>>(new Mensagem(e));
      }
   }
}
```

A mesma classe já vem com os métodos Listar, Alterar, Incluir e Excluir montado com o padrão e com referência para a classe TOTabela1. Alterações podem ser feitas conforme necessidade.



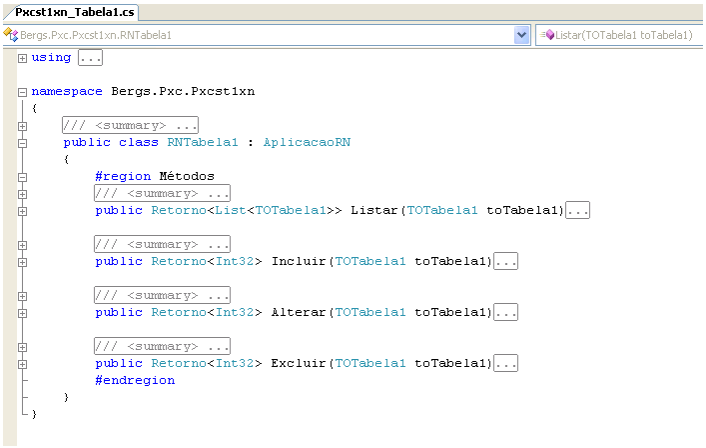
A classe AplicacaoBD dispõe dos métodos:

- Consultar: retorna uma lista de linhas da tabela onde cada linha possui uma coleção de campos

- Executar: executa um comando no banco de dados e retorna o número de linhas afetadas, que será 0 ou 1, se mais de um registro for afetado pelo comando, retorna a exceção MultiplosDadosException.

- ExecutarMultiplosDados: faz o mesmo que o comando Executar, porém permite que mais de um registro seja afetado.

A utilização da camada de banco de dados só pode ser feita a partir da camada de regras de negócios. Para isso, uma classe deve herdar de AplicacaoRN. O gerador de classes já gera a classe RN com os métodos padrões e a mesma deve ser alterada conforme a necessidade. Um exemplo segue abaixo.



Para realizar uma alteração no banco de dados, não podemos esquecer de iniciar a transação e efetivar, para isso utilizamos o EscopoTransacional, responsável por controlar a transação. O comando EfetivarTransacao realiza o “commit” no banco de dados.

```
public class RNTabela1 : AplicacaoRN
{
   //...
   //...
   public Retorno<Int32> Incluir(TOTabela1 toTabela1)
   {
      try
      {
         #region Validação de campos obrigatórios
         if (!toTabela1.Codigo.TemConteudo)
            return this.RetornarFalha<Int32>(new CampoObrigatorioMensagem("CODIGO"));
         if (!toTabela1.Data.TemConteudo)
            return this.RetornarFalha<Int32>(new CampoObrigatorioMensagem("DATA"));
         if (!toTabela1.Texto.TemConteudo)
            return this.RetornarFalha<Int32>(new CampoObrigatorioMensagem("TEXTO"));
         if (!toTabela1.Valor.TemConteudo)
            return this.RetornarFalha<Int32>(new CampoObrigatorioMensagem("VALOR"));
         #endregion
         //TODO: regras de negócio
         BDTabela1 bdTabela1 = this.InstanciarBD<BDTabela1>();
         Retorno<Int32> retIncluir;
         using (EscopoTransacional escopo = this.CriarEscopoTransacional())
         {
            retIncluir = bdTabela1.Incluir(toTabela1);
```

```
                if (!retIncluir.Ok)
                    return this.RetornarFalha<Int32>(retIncluir.Mensagem);
                escopo.EfetivarTransacao();
            }
            return this.RetornarSucesso<Int32>(retIncluir.Dados, new
OperacaoRealizadaMensagem("Inclusão"));
        }
        catch (Exception e)
        {
            return this.RetornarFalha<Int32>(new Mensagem(e));
        }
    }
}
```

Caso dê erro durante a execução do comando no bloco "try", automaticamente o roolback será executado pelo acionamento do "Dispose" do escopo transacional.

O método EfetivarTransacao() realiza o comando de commit no banco de dados.

Para interagir com a "ConsoleApplication" devemos ter uma classe que herde de AplicacaoTela. Abaixo segue um exemplo de como fazer.

```
class Program
{
    static void Main(string[] args)
    {
        using (MinhaTela minhaTela = new MinhaTela(@"C:\soft\pxc\data\Pxcz01da.mdb"))
        {
            minhaTela.Executar();
        }
    }
}
class MinhaTela : AplicacaoTela
{
    public MinhaTela(String caminho)
        : base(caminho)
    { }
    public void Executar()
    {
        //...
        try
        {
            //...
            Menu menu = new Menu(
                    new ItemMenu[] {
                        new ItemMenu( new KeyValuePair<int,string>(1, "Listar"), Listar, false),
                        new ItemMenu( new KeyValuePair<int,string>(2, "Incluir"), Incluir, false),
                        new ItemMenu( new KeyValuePair<int,string>(3, "Alterar"), Alterar, false),
                        new ItemMenu( new KeyValuePair<int,string>(4, "Excluir"), Excluir, false),
                        new ItemMenu( new KeyValuePair<int,string>(0, "Sair"), null, true)
                    }, null);
            Int32 op = Tela.ControlaMenu("Exemplo de utilização da classe MENU e acesso ao Access",
menu);
            if (op == 0)
            {
                Console.WriteLine("Saiu pela opção 0");
                Console.ReadKey();
            }
        }
        catch (Exception e)
        {
            Console.Write("Erro {0}\nTecle algo...", e.Message);
            Console.ReadKey();
        }
    }
```

```
void Incluir(object obj)
{
   try
   {
      RNTabela1 rnTabela1 = this.InstanciarRN<RNTabela1>();
      TOTabela1 toTabela1 = new TOTabela1();

      toTabela1.Codigo = Tela.Ler<Int32>("Informe o código: ");
      toTabela1.Texto = Tela.Ler<String>("Informe o texto: ");
      toTabela1.Data = Tela.Ler<DateTime>("Informe a data: ");
      toTabela1.Valor = Tela.Ler<Double>("Informe o valor: ");
      Retorno<Int32> retIncluir = rnTabela1.Incluir(toTabela1);
      if (!retIncluir.Ok)
      {
         Console.WriteLine("Erro na inclusão: {0}", retIncluir.Mensagem);
      }
      else
      {
         Console.WriteLine("Registros incluídos: {0}", retIncluir.Dados);
      }
   }
   catch (Exception e)
   {
      Console.Write("Erro {0}", e.Message);
      Console.ReadKey();
   }
}
void Listar(object obj)
{
   ImprimeLista("Lista\n");
}

String ImprimeLista(String titulo)
{
   RNTabela1 rnTabela1 = this.InstanciarRN<RNTabela1>();
   TOTabela1 toTeste = new TOTabela1();
   Int32 itemSelecionado = -1;

   Retorno<List<TOTabela1>> retListar = rnTabela1.Listar(toTeste);
   if (!retListar.Ok)
   {
      Console.WriteLine(retListar.Mensagem);
   }
   else
   {
      CabecalhoLista[] cabecalho = new CabecalhoLista[4];
      cabecalho[0] = new CabecalhoLista("Código", CabecalhoLista.AlinhamentoCelula.Direita);
      cabecalho[1] = new CabecalhoLista("Texto", CabecalhoLista.AlinhamentoCelula.Centralizado);
      cabecalho[2] = new CabecalhoLista("Data");
      cabecalho[3] = new CabecalhoLista("Valor", CabecalhoLista.AlinhamentoCelula.Direita);
      List<LinhaLista> registros = new List<LinhaLista>();
      foreach (TOTabela1 teste in retListar.Dados)
      {
         LinhaLista linha = new LinhaLista();
         linha.Celulas.Add(teste.Codigo.ToString());
         linha.Celulas.Add(teste.Texto.LerConteudoOuPadrao());
         linha.Celulas.Add(teste.Data.LerConteudoOuPadrao().ToString("dd/MM/yyyy"));
         linha.Celulas.Add(teste.Valor.LerConteudoOuPadrao().ToString("N"));
         registros.Add(linha);
      }
      itemSelecionado = Tela.ImprimeLista(titulo, cabecalho, registros, 15);
      if (itemSelecionado >= 0)
         return registros[itemSelecionado].Celulas[0];
   }
   return String.Empty;
}
```

```
        void Alterar(object obj)
        {
            RNTabela1 rnTabela1 = this.InstanciarRN<RNTabela1>();
            TOTabela1 toTeste = new TOTabela1();

            String codigoSelecionado = ImprimeLista("Selecione um item da lista e tecle ENTER para
alterar");
            if (!String.IsNullOrEmpty(codigoSelecionado))
            {
                toTeste.Codigo = Convert.ToInt32(codigoSelecionado);
                toTeste.Texto = Tela.Ler<String>("Informe o texto: ");
                if (Tela.Confirma("Deseja informar o valor? "))
                    toTeste.Valor = Tela.Ler<double>("Informe o valor: ");
                if (Tela.Confirma("Deseja informar a data? "))
                    toTeste.Data = Tela.Ler<DateTime>("Informe a data: ");
                Retorno<Int32> retAlterar = rnTabela1.Alterar(toTeste);
                if (!retAlterar.Ok)
                {
                    Console.WriteLine("Erro na alteração: {0}", retAlterar.Mensagem);
                }
                else
                    Console.WriteLine("Registros alterados: {0}", retAlterar.Dados);
            }
        }
        void Excluir(object obj)
        {
            RNTabela1 rnTabela1 = this.InstanciarRN<RNTabela1>();
            TOTabela1 toTeste = new TOTabela1();

            String codigoSelecionado = ImprimeLista("Selecione um item da lista e tecle ENTER para
excluir");
            if (!String.IsNullOrEmpty(codigoSelecionado))
            {
                if (Tela.Confirma(String.Format("Confirma a exclusão do código {0}? ", codigoSelecionado)))
                {
                    toTeste.Codigo = Convert.ToInt32(codigoSelecionado);
                    Retorno<Int32> retExcluir = rnTabela1.Excluir(toTeste);
                    if (!retExcluir.Ok)
                    {
                        Console.WriteLine("Erro na exclusão: {0}", retExcluir.Mensagem);
                    }
                    else
                    {
                        Console.WriteLine("Registros excluídos: {0}", retExcluir.Dados);
                    }
                }
            }
        }
    }
```