

# Treinamento em C#

# Sumário

1.Construindo Aplicações em C# .....	12
Processo de construção e compilação.....	13
O .NET Framework SDK .....	13
Exercício Prático 1 – Analisando as ferramentas do Framework.....	13
Sua primeira aplicação C# .....	14
O compilador csc.exe .....	14
Exercício Prático 2 – Configurando o ambiente.....	14
Exercício Prático 3 – Criando seu primeiro programa C#.....	15
Exercício Prático 4 – Compilando seu programa C# .....	16
O Visual Studio.NET 2005.....	17
Editor de Código e Interface.....	17
Web Server Integrado .....	18
.....	
Solution Explorer .....	18
Task List .....	19
Object Browser .....	19
.....	20
Dynamic Help .....	20
.....	21
Properties .....	21
Server Explorer.....	22
Class View .....	23
.....	24
Refactoring .....	24
.....	24
Alinhamento de Janelas .....	24
ToolBox .....	25
Sistema de Exibição de Páginas .....	26
Code Snippets .....	28

IntelliSense .....	29
Regiões de ocultação .....	29
Comentários XML.....	31
Breakpoints .....	31
Exercício Prático 5 – Criando um projeto no Visual Studio .....	32
2.Fundamentos da Linguagem C# .....	33
Estrutura de um programa C# simples .....	34
Utilizando a classe System.Environment .....	35
Exercício Prático 1 – Construindo uma aplicação C# .....	36
Declarando variáveis .....	36
Atribuição de valores e inicialização de variáveis .....	38
Utilizando operadores.....	39
Exercício Prático 2 – Utilizando variáveis e operadores.....	40
Conversão de dados .....	41
Conversão Explícita.....	41
Conversão Implícita.....	41
Conversões para strings .....	43
Convertendo valores para string .....	43
Convertendo string para outros valores .....	44
Exercício Prático 3 – Utilizando funções de conversão .....	44
Definindo classes e criando objetos.....	45
Value Type e Reference Types.....	46
Criando métodos em uma classe .....	46
Os modificadores in/out/ref .....	47
O modificador ref .....	48
O modificador out.....	48
Criando Propriedades .....	49
Exercício Prático 4 – Implementando classes .....	50
Utilizando Construtores.....	51
Definindo Constantes .....	52
Exercício Prático 5 – Criando construtores e utilizando constantes.....	52

Construindo Enumeradores .....	52
Exercício 6 – Criando enumeradores.....	53
Trabalhando com Arrays .....	54
Acessando elementos do array .....	54
Redimensionando Arrays .....	55
Classes de coleções.....	55
Exercício Prático 7 – Trabalhando com Arrays.....	55
Comandos de decisão (IF e SWITCH) .....	56
Exercício Prático 8 – Utilizando comandos de decisão.....	58
Comandos de Repetição (For/While/Foreach) .....	59
O comando For .....	59
O comando while.....	59
O comando foreach.....	60
Interrompendo a execução do loop.....	60
Exercício Prático 9 – Utilizando instruções de repetição .....	60
Definindo e utilizando namespaces .....	61
Exercício 10 – Criando Namespaces .....	62
Trabalhando com métodos estáticos.....	63
Exercício 11 – Criando métodos estáticos.....	63
Utilizando Strings do C# .....	64
O método Substring .....	64
O método IndexOf.....	64
Funções ToUpper e ToLower.....	64
Funções StartsWith e EndsWith.....	65
Funções TrimStart, TrimEnd e Trim .....	65
Funções PadLeft e PadRight.....	65
Funções String.Join e String.Split .....	65
A Função String.Format .....	66
A classe StringBuilder .....	67
Exercício Prático 12 – Utilizando operações de string.....	67
3.Utilizando o namespace System.IO.....	69
Introdução .....	70

O Namespace System.IO .....	70
Criando um arquivo texto .....	70
Mas afinal, o que é um Stream ? .....	71
Adicionando informações ao arquivo .....	71
Lendo arquivos binários .....	72
Abrindo arquivos em modos diferentes .....	72
Criando pastas no sistema de arquivos .....	73
Movendo e Copiando arquivos .....	73
Exercício Prático 1 – Lendo arquivos texto .....	73
Exercício Prático 2 – Gravando arquivos texto .....	74
Exercício Prático 3 – Alterando o arquivo de contas .....	74
4.Orientação à objetos no C# .....	75
Entendendo a estrutura de classes no C# .....	76
Encapsulamento .....	76
Abstração .....	76
Modificadores de acesso public e private .....	76
Herança .....	77
Exercício 1 – Implementando Herança .....	79
O modificador de acesso protected .....	79
Exercício Prático 2 – Implementando visibilidade protected .....	80
Implementando polimorfismo .....	81
O modificador virtual .....	81
O modificador override .....	82
A palavra reservada base .....	82
Regras de conversão de dados .....	83
A classe System.Object .....	83
Os métodos da classe Object .....	84
Exercício Prático 3 – Implementando Métodos virtuais .....	85
Classes parciais .....	85
Documentação XML .....	86
Exercício Prático 4 – Implementando documentação XML .....	87
5.Interfaces e coleções .....	88

Definindo interfaces no C#.....	89
Implementando interfaces no C#.....	90
Diferença entre interfaces e classes abstratas .....	91
Utilizando interfaces como parâmetros ou retorno de funções .....	91
As interfaces padrão do .NET Framework.....	91
A interface IEnumerable.....	92
A interface ICloneable.....	92
A interface IComparable .....	93
Exercício Prático 1 – Implementando Interfaces.....	94
6.Ciclo de vida dos objetos no C# .....	95
Classes, objetos e referências .....	96
Introdução ao ciclo de vida .....	96
O Garbage Collector.....	96
Finalização de objetos .....	97
Disposable Classes .....	97
Exercício Prático 1 – Implementando a interface IDisposable.....	98
7.Tratamento estruturado de exceções.....	100
A importância do tratamento estruturado de exceções .....	101
Exemplo simples de tratamento estruturado de exceções .....	101
System-Level Exceptions .....	102
Exercício Prático 1 – Utilizando tratamento estrutura de exceções .....	104
Application Exception .....	104
O bloco finally .....	105
Exercício Prático 2 – Utilizando exceções customizadas.....	106
8.Delegates e eventos .....	107
Introdução .....	108
Delegates .....	108
Chamando uma função sem delegates.....	108
Um delegate simples.....	108
Associando uma função estática a um delegate.....	109
Associando uma função de uma classe a um delegate .....	110
Apontando para mais de uma função .....	112

Exercício Prático 1 – Utilizando delegates .....	112
Eventos .....	113
Convenções utilizadas na criação de eventos .....	113
Exercício Prático 2 – Utilizando eventos .....	115
9.Técnicas avançadas de construção de tipos .....	117
Construindo tipos com indexadores .....	118
Overload de operadores .....	119
Rotinas de conversão .....	120
Exercício Prático 1 – Utilizando indexadores e overload de operadores .....	122
Comandos avançados no C#.....	122
Os comandos checked e unchecked.....	122
Exercício Prático 2 – Utilizando a diretiva checked.....	123
Diretivas de processamento .....	123
O atributo [Conditional] .....	125
Exercício Prático 3 – Compilação condicional.....	126
10.Entendendo Generics .....	127
Introdução .....	128
Revisando o conceito de boxing e unboxing .....	128
O namespace System.Collections.Generic.....	128
Analisando a classe List<T> .....	129
Exercício Prático 1 – Utilizando coleções genéricas.....	130
Criando classes e estruturas genéricas.....	131
Criando interfaces genéricas.....	133
Exercício Prático 2 – Implementando classes genéricas .....	134
11..NET Assemblys .....	135
Introdução .....	136
A reutilização de código e os assemblys .....	136
Criando um assembly .....	136
Alterando as propriedades dos assemblys .....	138
Referenciando um assembly .....	138
Exercício Prático 1 – Criando componentes.....	139
Private Assemblys e Shared Assemblys.....	140

Analisando as referencias .....	140
Entendendo a estrutura do GAC .....	141
Criando componentes compartilhados .....	141
Instalando um assembly no GAC .....	142
Binding policys .....	142
Exercício Prático 2 – Colocando assemblies no GAC .....	144
12.Reflection, Late Binding e Atributos .....	146
Introdução .....	147
Entendendo Reflection .....	147
A classe System.Type .....	147
Obtendo metadados através da classe Type .....	148
Exercício Prático 1 – Obtendo informações sobre tipos .....	149
Carregando assemblies dinamicamente .....	149
Entendendo Late Binding .....	150
A classe System.Activator .....	150
Chamando métodos sem parâmetros .....	150
Chamando métodos com parâmetros .....	151
Exercício Prático 2 – Chamando métodos dinamicamente .....	151
Trabalhando com atributos .....	152
13.Serialização .....	153
Entendendo a serialização de objetos .....	154
Configurando objetos para serialização .....	154
Utilizando o BinaryFormatter .....	154
Exercício Prático 1 – Serializando as classes de contas .....	155
Utilizando o SoapFormatter .....	155
Utilizando o XmlSerializer .....	156
Exercício Prático 1 – Utilizando serialização .....	157
14.Acessando dados com ADO.NET .....	158
O que é o ADO.NET ? .....	159
Os namespaces relacionados ao ADO.NET .....	159
O modelo de execução do ADO.NET .....	160
O modelo de execução em um ambiente conectado .....	160



O modelo de execução em um ambiente desconectado .....	161
Estabelecendo uma conexão com um banco de dados .....	161
Criando comandos .....	162
Executando comando.....	163
O método ExecuteNonQuery.....	163
O método ExecuteScalar .....	163
O método ExecuteReader .....	164
Passando parâmetros.....	164
Exercício Prático 1 – Executando comandos na base de dados .....	165
O que é um DataSet ? .....	165
O que é um DataAdapter ? .....	166
Criando um DataSet e um DataAdapter .....	166
Criando e preenchendo um DataSet .....	166
Exercício Prático 2 – Utilizando um DataSet.....	167
15.Introdução ao ASP.NET .....	168
O que é o ASP.NET ? .....	169
O que é um WebForm ?.....	169
Bibliotecas de classe utilizadas.....	170
Entendendo os WebForms.....	170
Exercício Prático 1 – Criando um projeto Web.....	171
O que são controles Web ? .....	171
A estrutura de um controle Web.....	172
A referência no servidor .....	172
Propriedades dos controles web .....	173
WebServer Controls .....	173
Html Server Controls.....	173
Exercício Prático 2 – Criando uma aplicação ASP.NET.....	174
16.Construindo aplicações em ASP.NET.....	175
Entendeno o gerenciamento de estado.....	176
Gerenciando o estado da página .....	176
Interagindo com o ViewState .....	176
O que são Cookies ? .....	176

Criando cookies temporários .....	177
Lendo informações de um Cookie.....	177
O que são variáveis de Sessão ? .....	177
Como funciona o gerenciamento de Sessão ? .....	178
Interagindo com variáveis de sessão .....	178
O controle de sessão .....	178
Modos de armazenamento de sessão .....	179
Considerações sobre escalabilidade.....	179
Iniciando o StateServer.....	179
Variáveis de aplicação .....	179
O arquivo global.asax .....	180
Eventos do global.asax .....	180
Exercício Prático 1 – Utilizando cookies, variáveis de sessão e aplicação .....	180
O modelo de configuração do ASP.NET.....	181
O arquivo Machine.Config .....	181
O arquivo Web.Config .....	181
Hierarquia de configurações .....	182
Criando itens de configuração .....	182
Lendo informações de configuração.....	182
Exercício Prático 2 – Os recursos de configuração do ASP.NET .....	183
17.Construindo Xml Web Services .....	184
O que são web services ?.....	185
Como o .NET torna os web services possíveis ?.....	185
Como funciona a comunicação ?.....	185
Como funcionam os web services ? .....	185
Utilizando um web service .....	186
Chamando um web service através de uma URL.....	186
Utilizando uma classe Proxy .....	186
Adicionando uma referencia web .....	186
Instanciando uma classe Proxy.....	186
Exercício Prático 1 – Utilizando um web service.....	187
Criando um web service .....	187

Criando métodos.....	187
Exercício Prático 2 – Criando um web service.....	188

# **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# na 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 exista a opção de não utilizar uma IDE de desenvolvimento, sua instalação é totalmente recomendada.

Uma série de ferramentas acompanha o .NET Framework Sdk. Estas 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.

## ***Exercício Prático 1 – Analisando as ferramentas do Framework***

1. Abra o Windows Explorer
2. Localize a pasta de instalação do .NET Framework SDK
3. Localize o arquivo StartHere.HTM e abra este arquivo utilizando o Internet Explorer
4. Localize na página de apresentação do SDK a a região “Tools and Debuggers”

5. Clique no link “tools and debuggers”
6. Verifique as ferramentas disponíveis no .NET Framework SDK

## ***Sua primeira aplicação C#***

Conforme já comentamos anteriormente, os programas C# podem ser desenvolvidos utilizando um simples editor de texto, como o notepad. Os arquivos de código do C# possuem a extensão .cs, apesar de o uso desta extensão não ser mandatório. Um programa C# pode ser composto por diversos arquivos .CS, que quando compilados se tornam um módulo único.

## ***O compilador csc.exe***

Tendo um arquivo com código fonte .cs, para que este arquivo se torne efetivamente um programa, precisamos que este arquivo seja compilado. Caso estejamos trabalhando apenas com o .NET Framework SDK, temos que trabalhar com o compilador de linha de comando, o csc.exe. É lógico que você não irá criar os seus programas utilizando esta ferramenta, mas o conhecimento dela pode ser útil em situações em que a IDE de desenvolvimento não esteja disponível. Outro ponto importante é que a ferramenta de desenvolvimento na realidade utiliza o compilador de linha de comando para efetuar a compilação dos programas.

O compilador do C# possui a seguinte sintaxe.

```
Csc /out:<arquivo de saída> /target:[exelib] <arquivo.cs> [<arquivo.cs>] [<arquivo.cs>] [...]
```

O parâmetro out indica o nome do arquivo compilado que será gerado (em geral um arquivo com a extensão .DLL ou .EXE). A opção target indica o tipo de arquivo (exe para executável e lib para arquivos dll).

## ***Exercício Prático 2 – Configurando o ambiente***

1. Na área de trabalho, clique com o botão direito sobre o ícone “Meu Computador” e selecione a opção “Propriedade”
2. Clique na aba “Avançado”
3. Clique no botão variáveis de ambiente

4. Localize a variável de sistema “Path”
5. ALTERE a variável para incluir um NOVO caminho, que será o caminho para o .NET Framework. Você deve adicionar ao final do valor existente na variável um símbolo ponto e vírgula (“;”) e o seguinte caminho

```
C:\Windows\Microsoft.NET\Framework\v2.0.50727
```

6. Clique em para confirmar fechar a janela de propriedade do meu computador
7. Inicie o prompt de comando (Iniciar, Programas, Acessórios, Prompt de Comando)
8. Digite o comandos

```
csc /?
```

9. Verifique que as opções do compilador C# são apresentadas

### ***Exercício Prático 3 – Criando seu primeiro programa C#***

1. Inicie o Notepad
2. Digite o seguinte conteúdo no arquivo

```
using System;
namespace Modulo01
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Bem vindo ao C#;");
        }
    }
}
```

Salve o arquivo na pasta C:\CSharp\Modulo01 com o nome de Modulo01.cs

### ***Exercício Prático 4 – Compilando seu programa C#***

1. Inicie o prompt de comando
2. Desvia para pasta C:\CSharp\Modulo01
3. Utilizando a linha de comando a seguir, compile seu programa

```
csc /out:Modulo01.exe /target:exe Modulo01.cs
```

4. Verifique que o arquivo Modulo01.exe foi criado. Caso não tenha sido criado, verifique se você digitou corretamente a estrutura do programa
5. Execute o programa digitando Modulo01 no prompt de comando



## ***O Visual Studio.NET 2005***

É lógico que a utilização do notepad 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 uma 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***

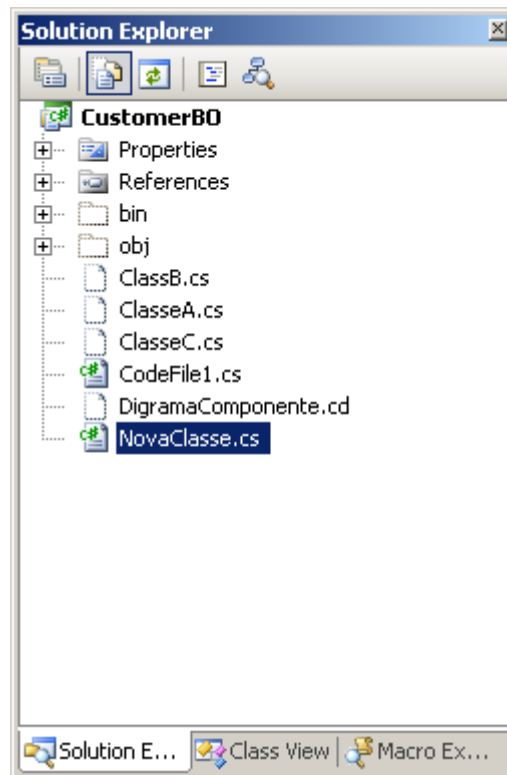
O Visual Studio 2005 traz integrado a seu sistema, um web server totalmente desvinculado ao IIS. Agora, aqueles que não tem IIS instalado em seus computadores, podem utilizar o Visual Studio 2005 para criar seus sites e web services e executá-los facilmente.

O web server integrado, é seguro pois não permite acesso por parte de outras máquinas (somente a máquina que está 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.



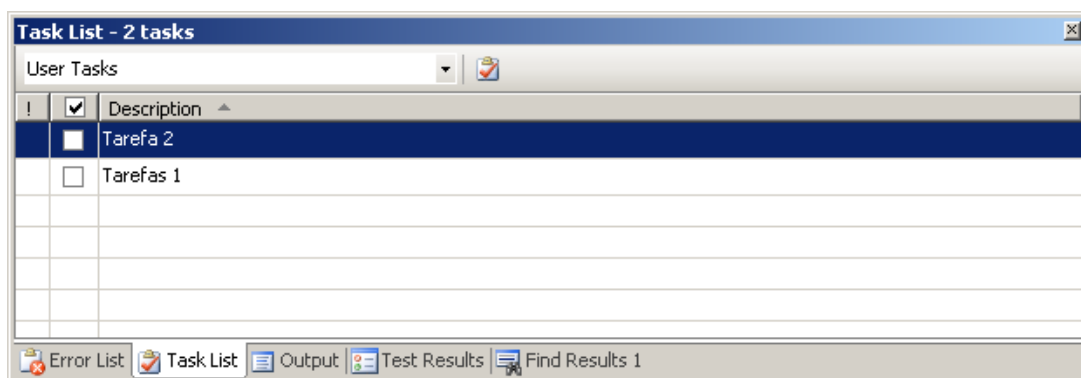
## Task List

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.

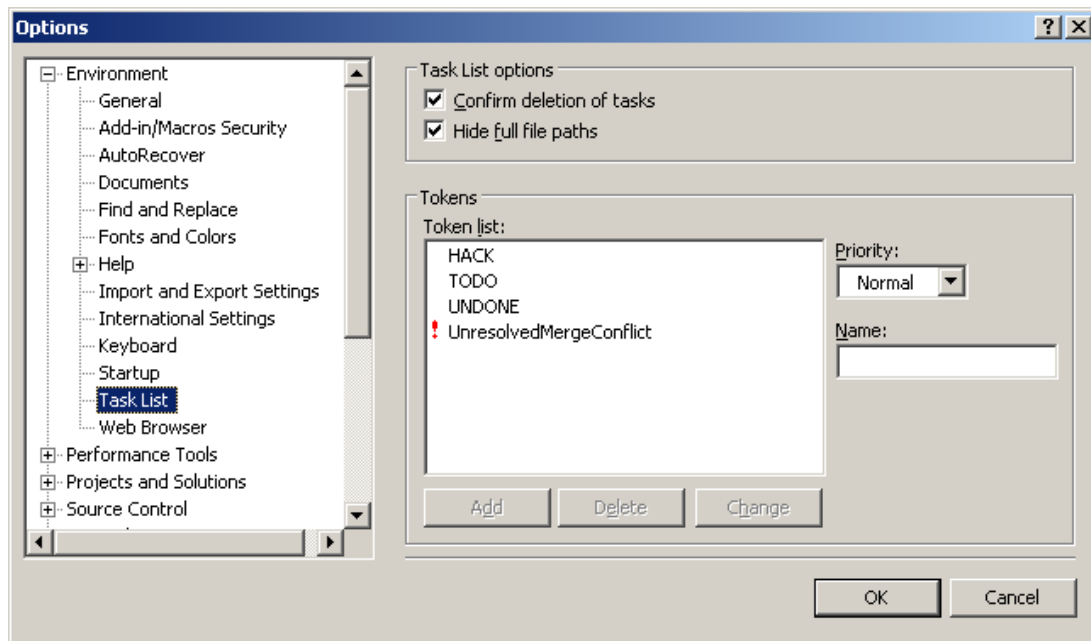
Para inserirmos algum item falante na programação, podemos incluir por exemplo um comentário com a marca TODO, como o exemplo a seguir.

`//TODO: Implementar o código para validação do usuário`

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



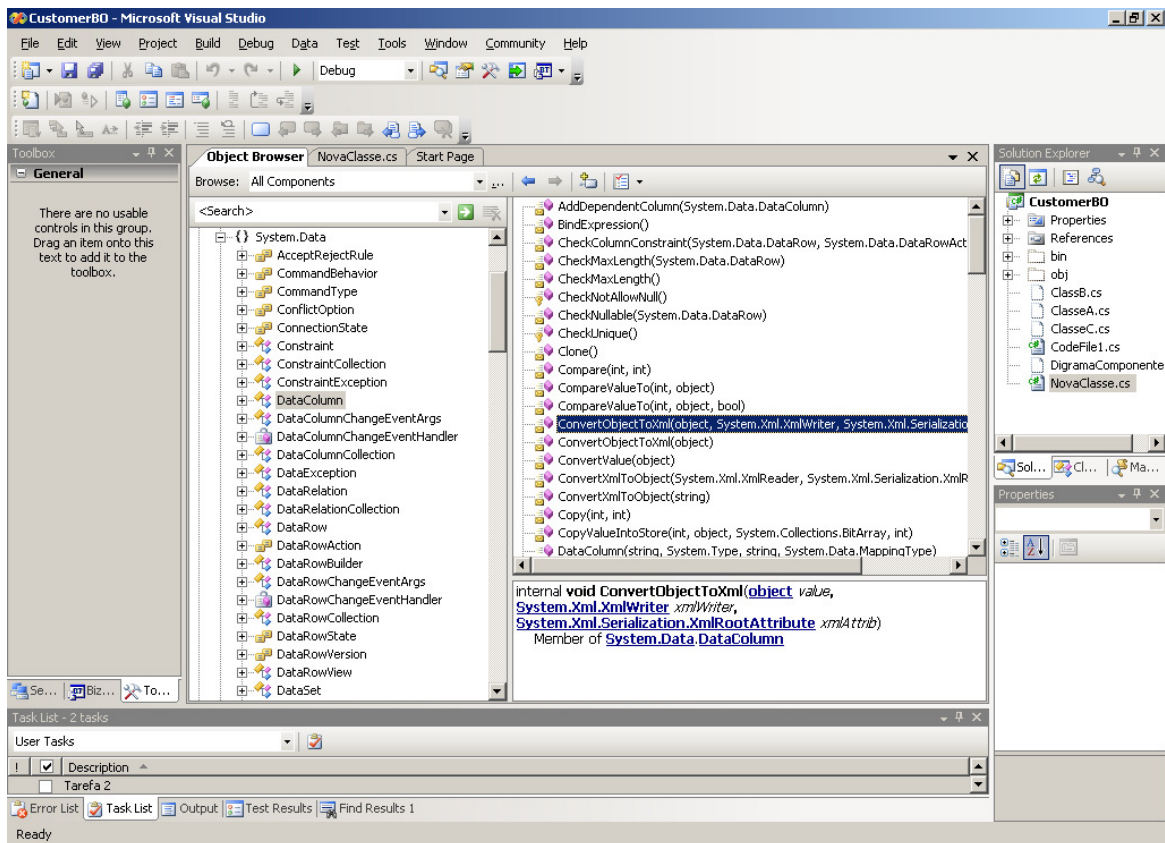
Os tokens utilizados para a criação de tarefas através de comentários podem ser configurados através do menu Tools, opção Options, na área “Environment”/”Task List”.



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

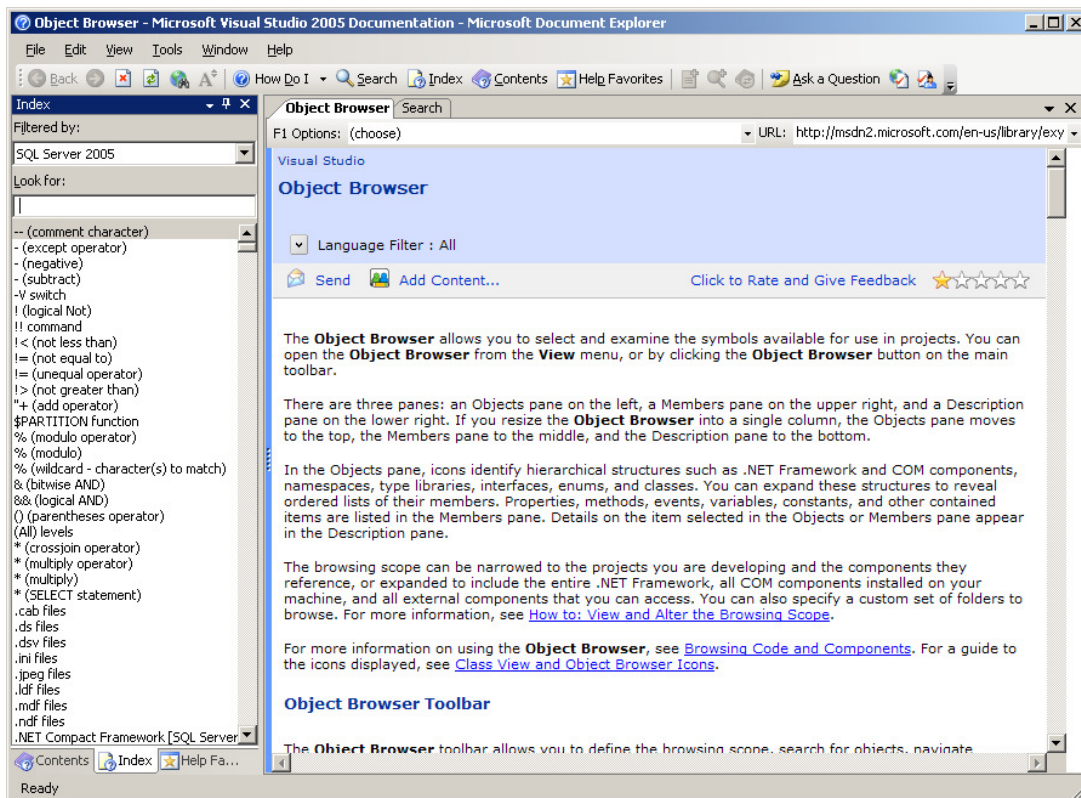
## ***Object Browser***

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”



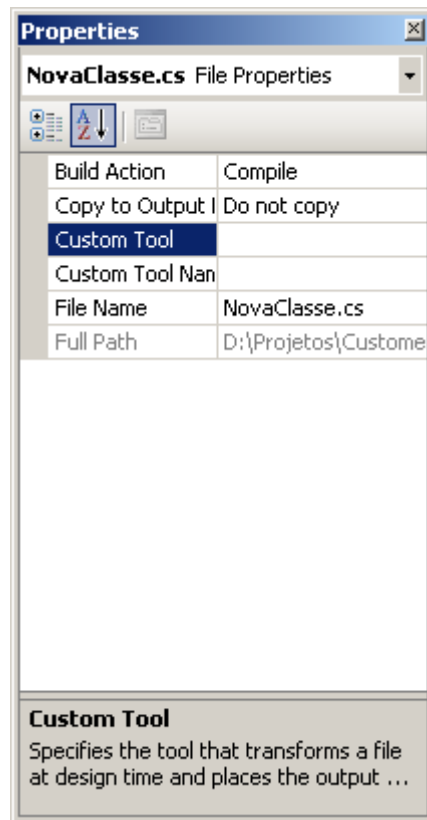
## Dynamic Help

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.



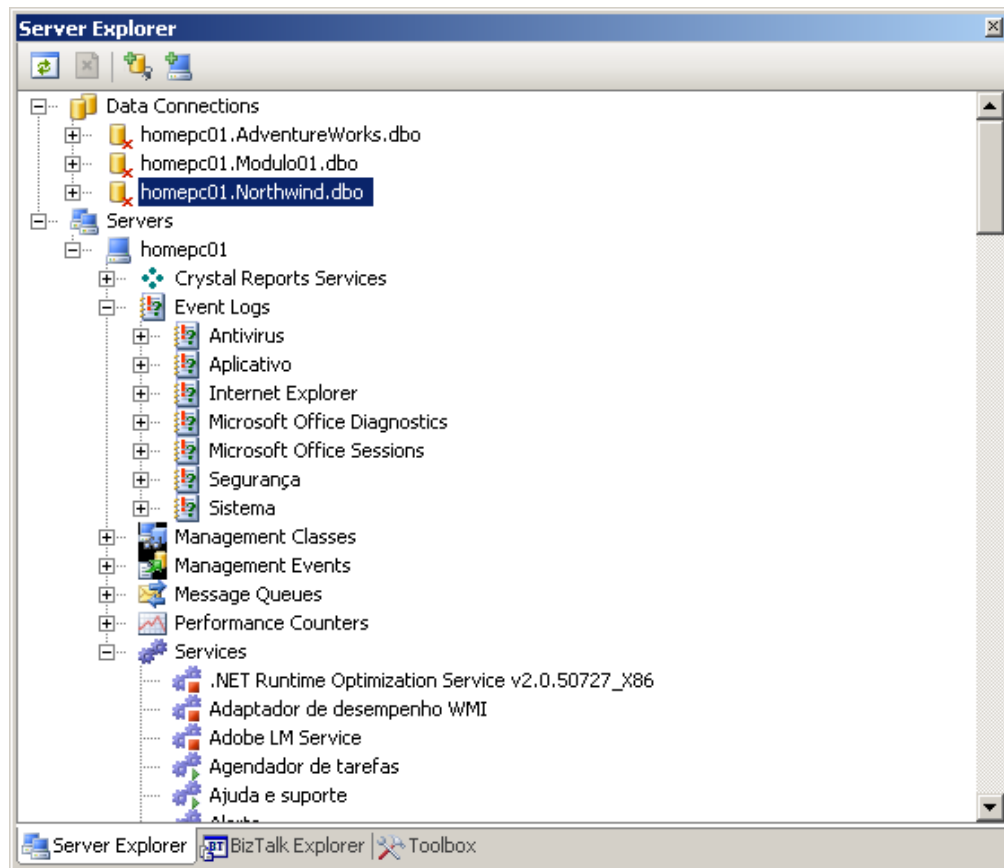
## Properties

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***

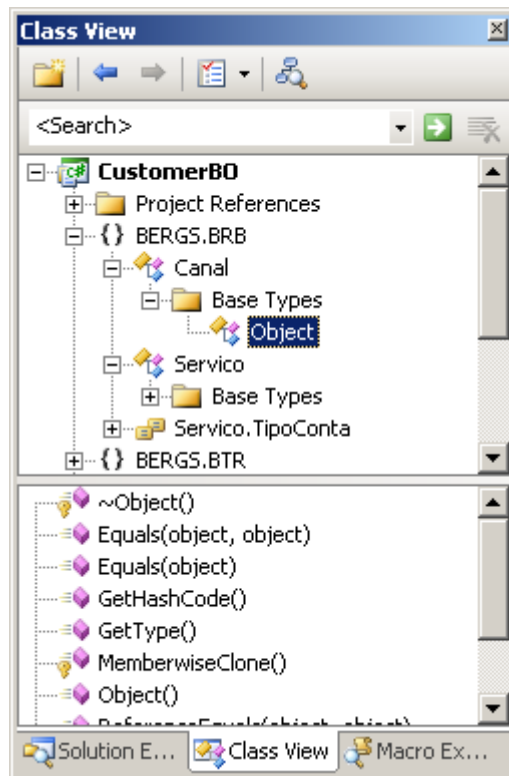
O Server explorer permite o acesso a recursos de servidores que podem ser utilizados durante o desenvolvimento da aplicação. Através desta janela é possível iniciar e parar serviços de um servidor, ver os bancos de dados disponíveis, a log de eventos e outros recursos adicionais. Também é possível a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



## ***Class View***

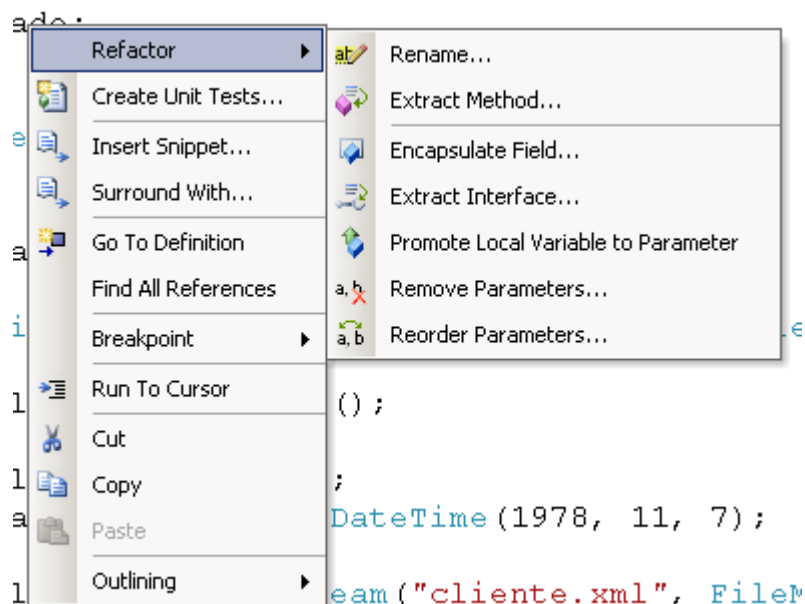
A janela Class View apresenta uma visualização dos tipos definidos no seu projeto de forma hierárquica (namespaces, classes, heranças, etc.). Um quadro de combinação (combo box) permite a busca por tipos de modo rápido e simples. A parte inferior apresenta os membros do tipo selecionado na parte superior,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





## Refactoring

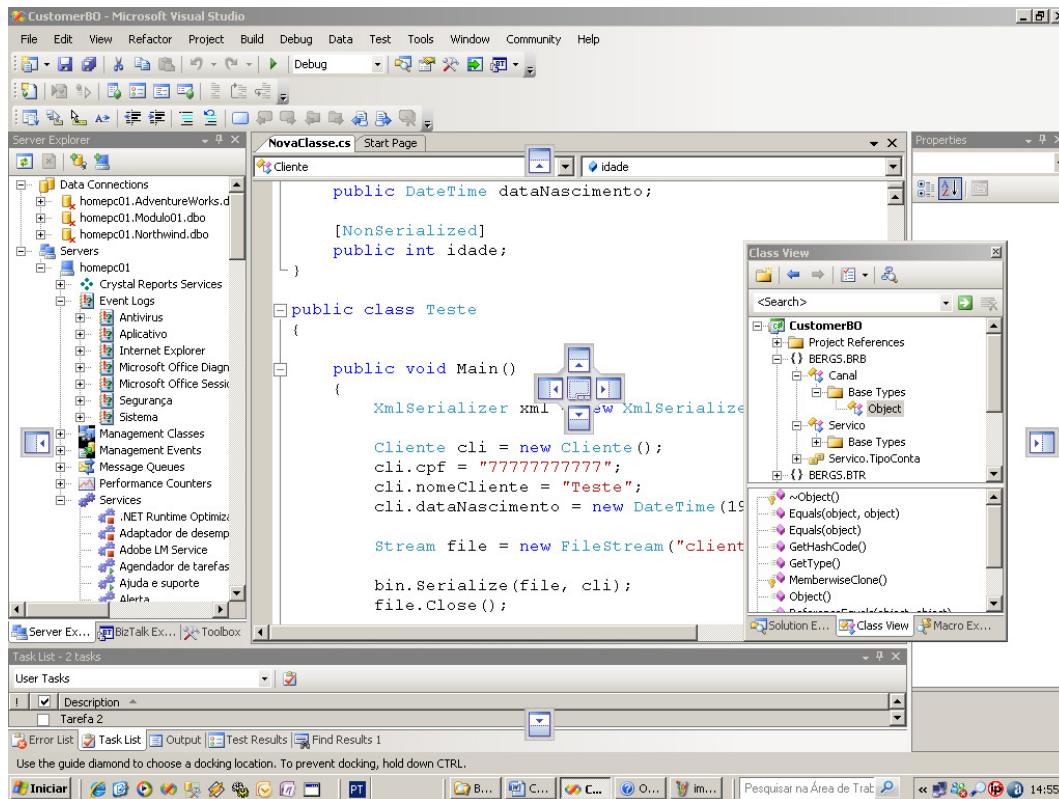
Além das janelas já apresentadas aqui, o Visual Studio ainda possui uma série de recursos que facilitam a edição e manutenção do código fonte dos programas. Este recurso também é conhecido por refactoring, e permite a troca de nome de métodos ou de variáveis e a replicação desta mudança em todo o código 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

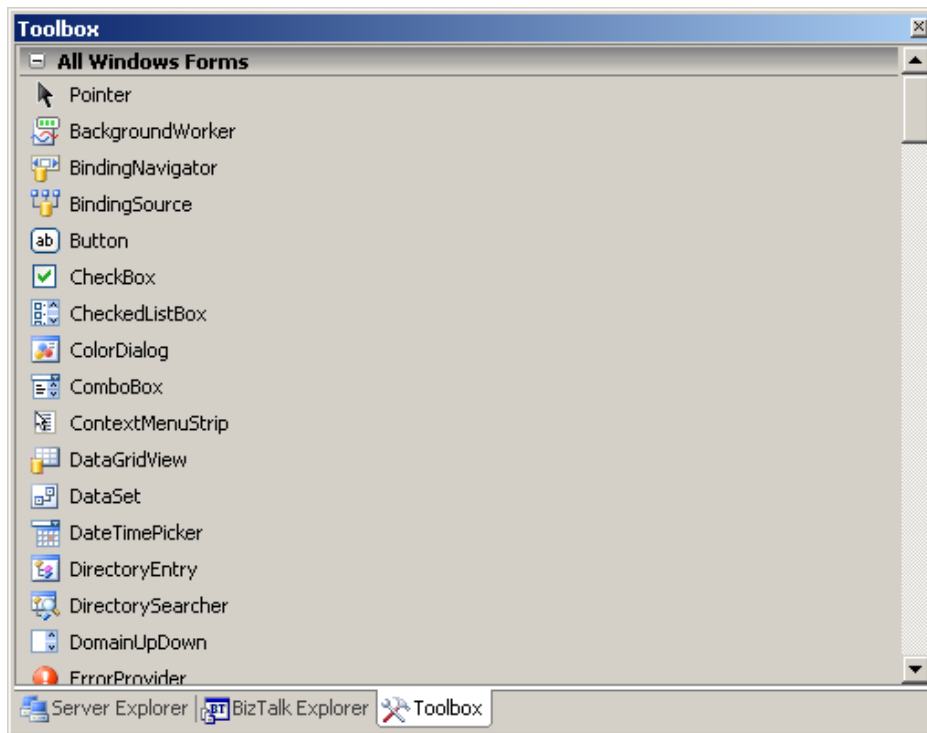
(Solution Explorer, Properties e etc.) dentro do IDE. Agora, quando arrastamos uma janela para fora da estrutura padrão, fica muito simples arrumar o layout da maneira escolhida. Para isso, basta que o desenvolvedor clique com o botão direito do mouse sobre a barra da Janela e selecione a opção Dockable. 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 alinhada a partir da janela sobreposta; Ao clicar no centro a janela será alinhada ao centro, enquanto os direcionais permitem que seja alinhada acima, abaixo, à esquerda ou à direita da janela sobreposta.

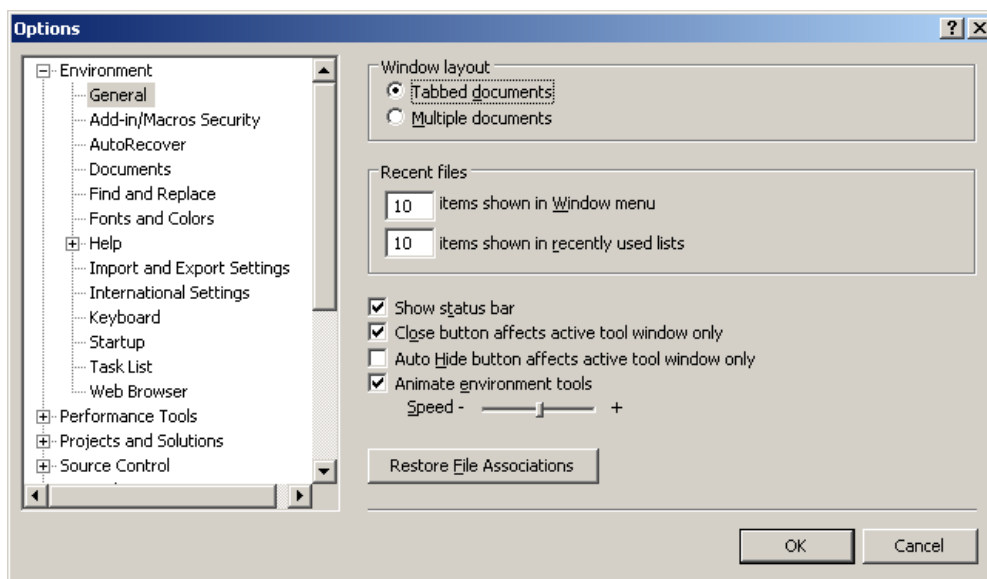
## ToolBox

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.

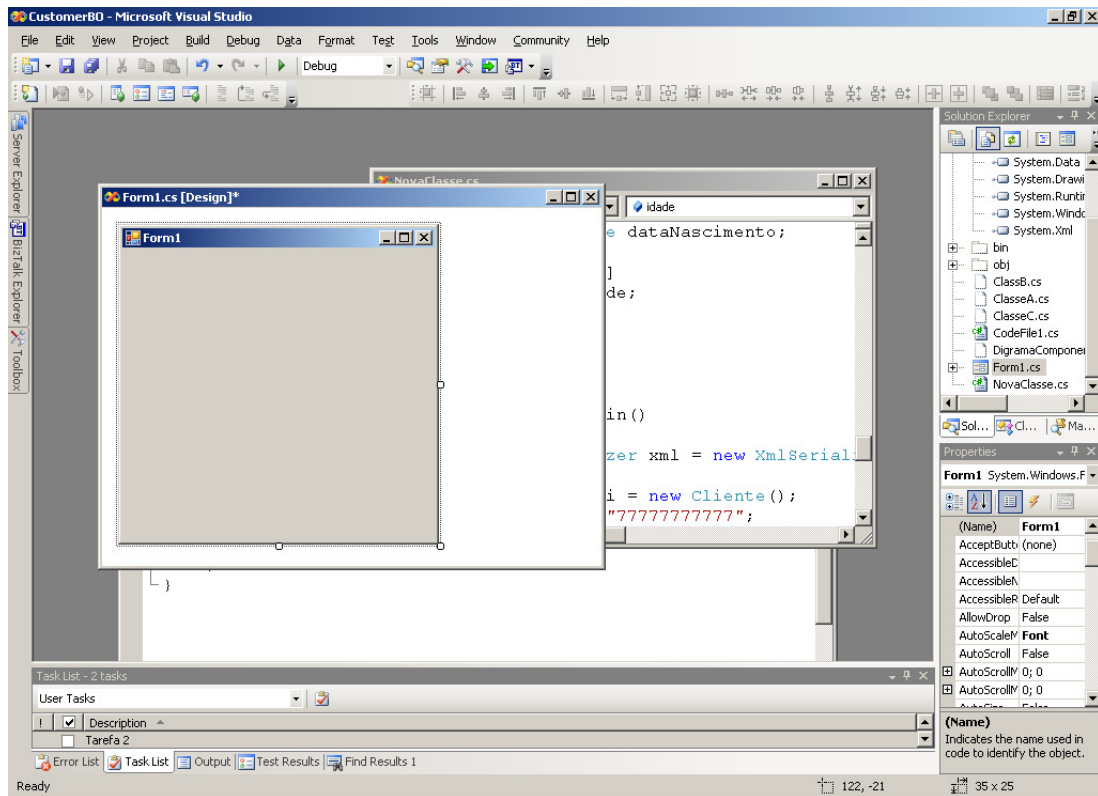


## ***Sistema de Exibição de Páginas***

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:



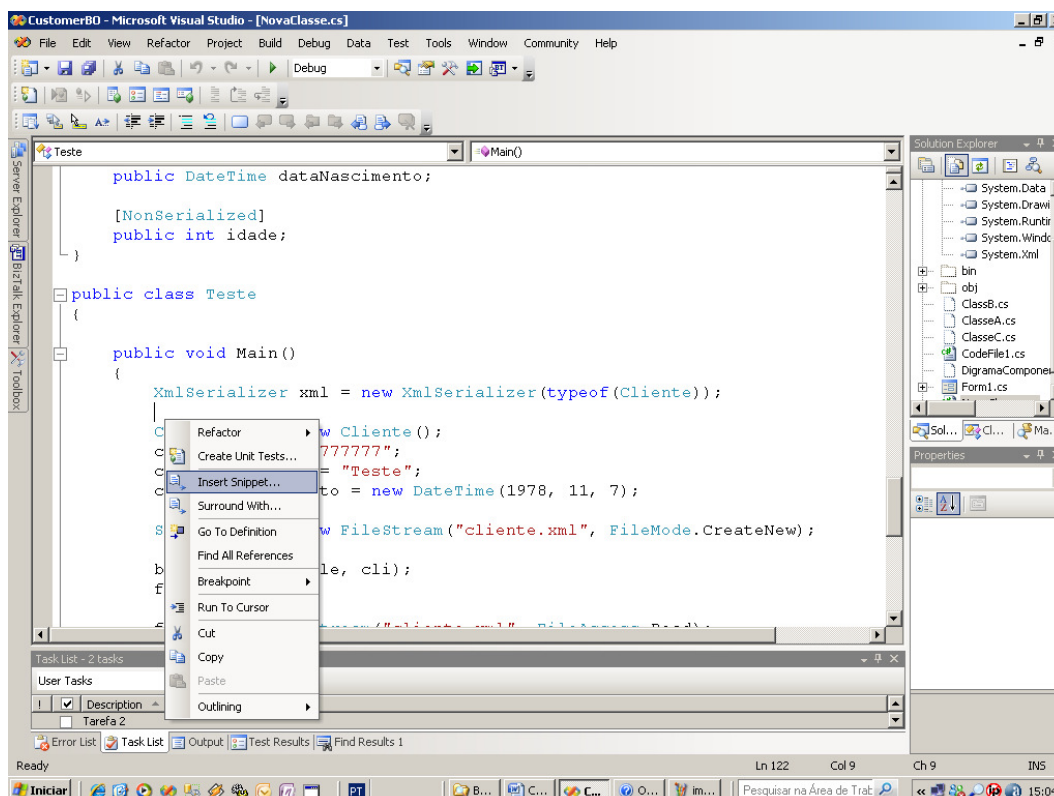
O modelo Multiple Documents é semelhante ao modelo utilizado pela versão anterior do Visual Basic.



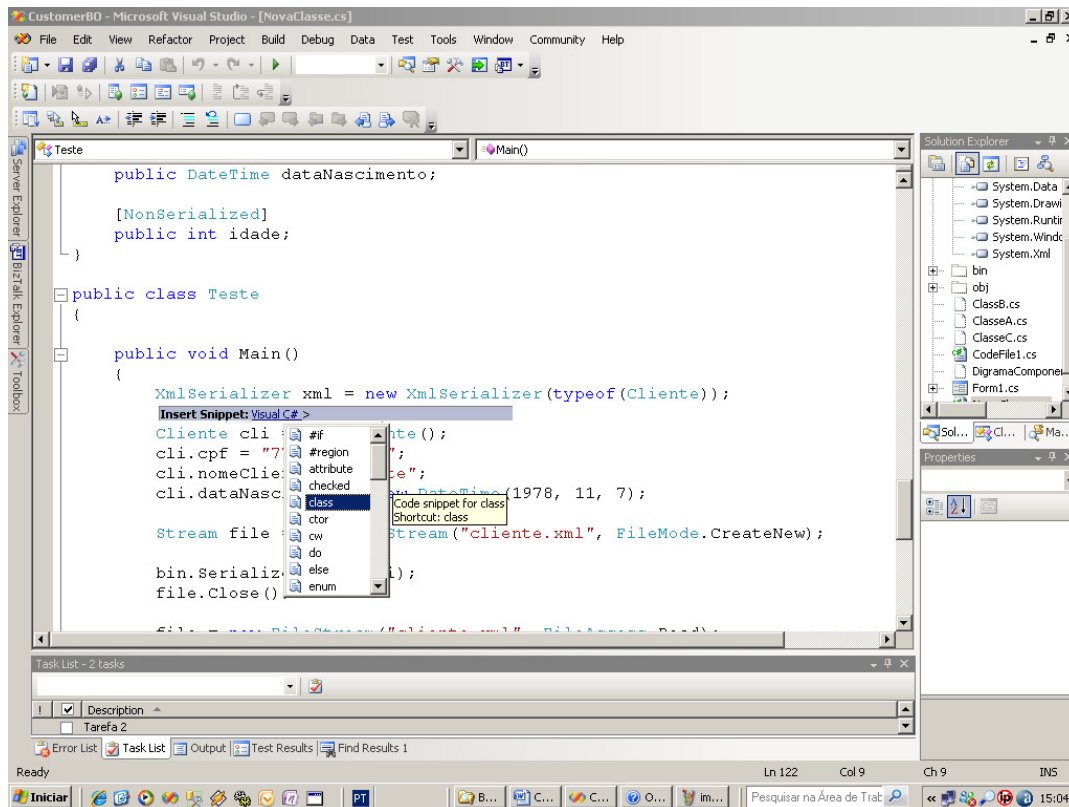
Os documentos abertos são exibidos como janelas flutuantes, permitindo ao desenvolvedor posicioná-las de acordo com suas necessidades.

## 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.

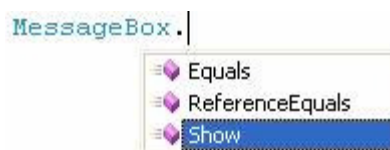


O desenvolvedor poderá escolher o tipo de código que será inserido no Code Editor.

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.

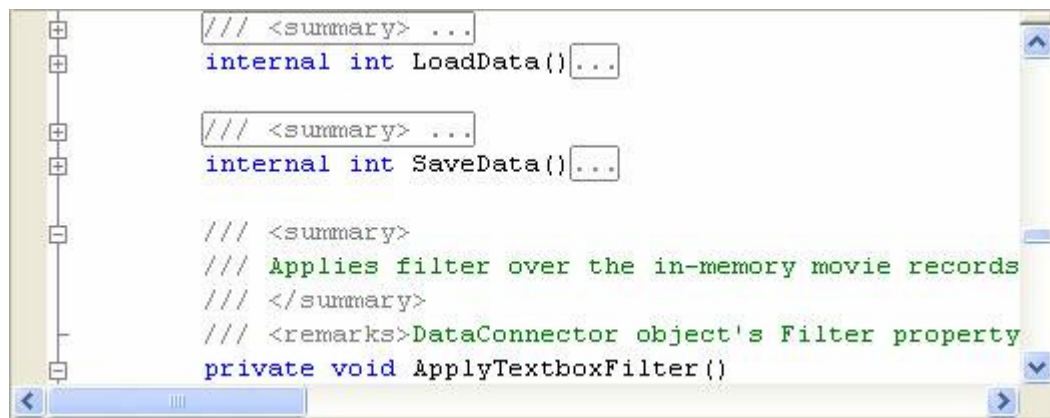
## IntelliSense

O intellisense 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 determinado contexto.



```
/// <summary> ...  
internal int LoadData() ...  
  
/// <summary> ...  
internal int SaveData() ...  
  
/// <summary>  
/// Applies filter over the in-memory movie records  
/// </summary>  
/// <remarks>DataConnector object's Filter property  
private void ApplyTextboxFilter()
```



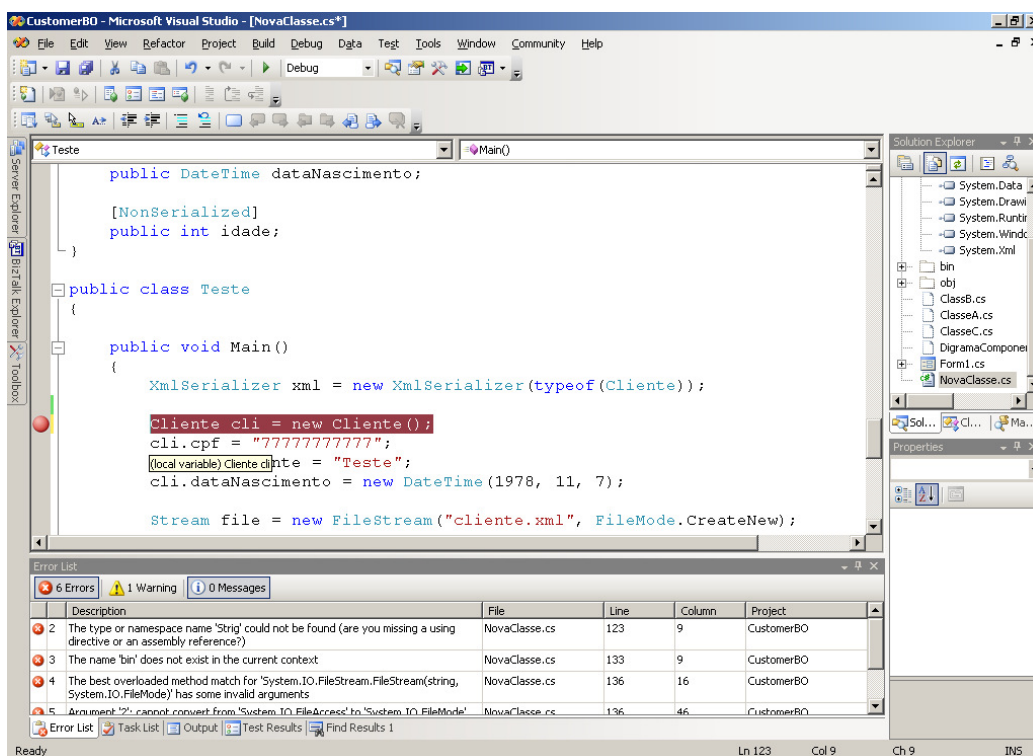
## Comentários XML

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 (///).

```
/// <summary>
/// Loads movie collection data.
/// </summary>
/// <returns>Positive integer for number of rows loaded; 0 :
/// <remarks>Loads movie collection data into the in-memory
internal int LoadData()
{
}
```

## Breakpoints

Os breakpoint definem linhas de parada para execução passo a passo em modo de depuração para análise de código.





## ***Exercício Prático 5 – Criando um projeto no Visual Studio***

1. Abra o Visual Studio.NET 2005
2. Crie um novo projeto com o nome de Modulo01 do tipo Console Application
3. Adicione o código do programa criado anteriormente com o notepad no arquivo class1.cs, disponível no solution explorer
4. Compile o programa através do menu build
5. Acesse a pasta onde foi criado o projeto e localize o executável gerado. Execute o programa e verifique o resultado
6. Adicione um novo comentário com o token “TODO:” e verifique se o comentário aparece na lista de tarefas
7. Acesse a janela class View e verifique os itens do seu projeto
8. Acesse a janela Object Browser e verifique os componentes do .NET disponíveis

## **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.

## ***Exercício Prático 1 – Construindo uma aplicação C#***

3. Inicie o Visual Studio.NET
4. Crie um novo projeto C# do tipo Console Application
5. Na função Main, implemente um código que exiba as seguintes informações para o usuário:
  - a. Nome da máquina
  - b. Domínio de rede do usuário logado
  - c. Código do usuário logado
  - d. Diretório do sistema
  - e. Número de processadores

## ***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.

<i>Common Type System Type</i>	<i>C#</i>	<i>Visual Basic.NET</i>
System.Byte	byte	Byte
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 = "Fabrício Scariot";
```

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 variáveis 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 dividido pelo operando

## ***Exercício Prático 2 – Utilizando variáveis e operadores***

1. Inicie o Visual Studio.NET
2. Crie um novo programa do tipo console application
3. Faça com que o programa exiba a seguinte mensagem ao usuário (substitua os valores dentre <> pelas informações correspondentes obtidas através da classe

Environment):

```
Bem-vindo <nome do usuário> agora são <hora atual>.  
Sua estação de trabalho é <nome da estação de trabalho>
```

## ***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 explícita 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 necessário, 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. As strings só podem ser convertidas 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 possui a função ToString e em alguns casos esta função pode inclusive receber parâmetros. Um exemplo de conversão 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 Prático 3 – Utilizando funções de conversão***

1. Inicie o Visual Studio.NET
2. Crie um novo programa do tipo Console Application
3. Utilizando uma variável string e a função `ReadLine` da classe `Console`, leia um valor da tela do usuário
4. Crie uma segunda variável string e leia um segundo valor do usuário
5. Converta os dois valores lidos para inteiro
6. Some os dois valores e mostre na tela sob forma de um string para o usuário.

## 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 instancia de um 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 contêm 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 contêm um valor, mas sim uma referência para uma área de memória que contém 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 as 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 do 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 cláusula 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 passado 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 utilizando os modificadores 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 continua 100, não é alterado  
    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 continua 100, não é alterado
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 armazenadas pelas classes. As propriedades não são simplesmente variáveis com visibilidade pública, mas sim estruturas de dados capazes de abstrair a complexidade de uma 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 a 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 armazenar 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;
        }
    }
}
```

### ***Exercício Prático 4 – Implementando classes***

1. Abra o Visual Studio, criando um novo projeto Console Application com o nome de Banco
2. Crie um novo arquivo de classe, com uma classe chamada ContaBancaria
3. Declare uma variável interna com o nome de m\_Saldo, do tipo Double
4. Implemente o método Depositar que receberá um valor Double e incrementará o saldo do cliente
5. Implemente o método Sacar que receberá um valor Double e retirará o valor informado do saldo do cliente

6. Crie uma propriedade somente com o modificador get chamada SaldoAtual
7. Crie outra propriedade chamada NumeroConta
8. Crie outra propriedade chamada NomeCorrentista
9. Na função Main do programa, crie uma nova instancia da classe ContaBancaria
10. Solicite ao usuário que informe o nome do correntista e o número da conta e atribua os respectivos valores a instância da classe
11. Solicite ao usuário que informe um valor para depósito em conta e efetue a chamada ao método Depositar
12. Solicite ao usuário que informe um valor para saque em conta e efetue a chamada ao método Sacar
13. Exiba ao usuário o saldo final da conta após as operações de depósito e saque

## ***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
```

## 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 Prático 5 – Criando construtores e utilizando constantes

1. Na classe ContaBancaria criada no exercício anterior, crie uma nova variável privada, do tipo Double com o nome de m\_LimiteCredito;
2. Declare uma constante com o nome de CPMF, do tipo Double com o valor de 0.0038;
3. Utilizando a constante, desconte o CPMF para cada saque feito através do método Sacar da classe ContaBancaria
4. Execute o programa novamente e verifique se o valor do CPMF está sendo descontado

## 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, utilizamos 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 correspondente a 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 atribuída 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 6 – Criando enumeradores***

1. No projeto criado no exercício anterior, crie um novo enumerador chamado TipoPessoa, contendo os valores Física e Jurídica
2. Altere a propriedade TipoPessoa da classe conta para que ela devolva o tipo do enum ao

invés de um string

3. Crie um novo enumerador chamado TipoConta. Informe os tipos de conta Poupança, Corrente e Investimento.
4. Altere a função Main para que seja solicitado o tipo de conta e o tipo de pessoa
5. Execute o programa e teste novamente

## ***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 as 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 array 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 a 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 automática. 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 há a necessidade de redimensionamento quando inserimos ou removemos um item. Nos próximos capítulos veremos mais detalhes sobre a utilização de lista utilizando Generics.



## ***Exercício Prático 7 – Trabalhando com Arrays***

1. Na classe Conta, implemente uma nova variável membro do tipo ArrayList, com o nome de operações
2. No construtor da classe, inicialize a variável do arraylist com o comando new
3. No método sacar, insira uma nova string lista indicando o saque do valor solicitado pelo usuário
4. No método depositar, insira uma nova string na lista indicando o depósito do valor informado pelo usuário
5. Execute o código e corrija os erros necessários
6. Verifique se os valores estão sendo inseridos no array em modo debug, utilizando um breakpoint

## ***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 à 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 quantos 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.

### ***Exercício Prático 8 – Utilizando comandos de decisão***

1. No exercício criado anteriormente, crie uma nova função chamada ValidarSaque, que retorne um valor do tipo booleano
2. Utilize a cláusula IF para verificar se o cliente possui saldo antes de efetuar um saque. Não se esqueça de validar o valor do saque contando o limite de crédito do cliente]
3. Insira a chamada da função ValidarSaque no método Sacar da classe
4. Crie uma nova propriedade na conta chamada de IsentoImpostos

5. Utilizando a cláusula IF, faça o teste para verificar se o CPMF deve ser descontado ou não do cliente
6. Crie uma nova classe dentro do mesmo arquivo com o nome de OperacaoContabil
7. Crie três variáveis membro na classe, com as seguintes informações
  - a. DataLancamento
  - b. Valor
  - c. Historico
8. Implemente um construtor na classe OperacaoContabil para atribuir os valores as variáveis criadas
9. Ao invés de inserir um texto no ArrayList, crie uma nova instancia da classe OperacaoContabil, com a data de hoje, o valor da operação e um histórico indicando saque ou depósito. Insira a instancia da classe OperacaoContabil no ArrayList

## ***Comandos de Repetição (For/While/Foreach)***

O C# possui três tipos de comando de repetição, for, foreach e while. Estes 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(variável=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 Prático 9 – Utilizando instruções de repetição***

1. Abra a solução criada no exercício anterior
2. Crie um novo método chamado Extrato que imprima na tela todas os lançamentos gerados na lista. Utilize o comando foreach para fazer a iteração.
3. Altere a função Main do programa para que ela implemente na tela de console uma tela que permite ao usuário as seguintes operações:
  - a. Sacar
  - b. Depositar
  - c. Extrato
  - d. Sair

Faça com que as opções sejam apresentadas ao usuário enquanto ele não selecionar a opção sair. Utilize o comando while para fazer esta operação. Antes de apresentar esta tela de opções, lembre-se de solicitar o nome do correntista, o número da conta e o tipo de pessoa

## ***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.
{
    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.
{
    public class Teste
    {
        public void subrotina()
        {
            Servico obj = new Servico();
        }
    }
}
```

```
}  
}
```

## ***Exercício 10 – Criando Namespaces***

1. Altere a classe Conta criada no exercício anterior para que ela faça parte do namespace BERGS..
2. No arquivo do programa de console, utilize a cláusula USING para importar o namespace BERGS.
3. Crie um novo arquivo de classe com a declaração para o namespace BERGS.BTR

## ***Trabalhando com métodos estáticos***

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

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 instancia 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 instancia 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");
```

## ***Exercício 11 – Criando métodos estáticos***

1. No arquivo com o namespace BERGS.BTR criado no exercício anterior, implemente uma nova classe chamada SistemaLog
2. Implemente um método estático que aceite um parâmetro do tipo string. Não é necessário implementar o método, a implementação será feita nos próximos exercícios.

## ***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 utilizados 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 = "Fabrício";  
  
nome = nome.ToUpper();  
nome = nome.ToLower();
```

## Funções StartsWith e EndsWith

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

```
string arquivo = "brbsglvw.dll";  
  
if(nome.StartsWith("brb"))  
    Console.WriteLine("É do sistema BRB!");  
  
if(nome.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 início e do fim da string.

```
string nome = " SCARIOT ";  
  
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 = "Scariot";

nome = nome.PadRight(10, ' ');
// "Scariot  "

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 poderam 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 substituido 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 à 2. A função também recebe 3 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:d/M/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);
```

## ***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.

```
//Codigo ineficiente  
string sql;  
sql = "SELECT * FROM Products";  
sql += " WHERE CategoryId=1 AND";  
sql += "     AND UnitPrice>10";  
  
//Codigo Eficiente  
StringBuilder sb =  
    new StringBuilder();
```

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

## ***Exercício Prático 12 – Utilizando operações de string***

1. Implemente uma função chamada `ImprimirSaldo` que retorne um string com a seguinte informação.  
  
Seu saldo atual é de R\$ XXXX,XX  
  
Utilize a função `String.Format` para obter esta string.
2. Altere o menu de opções da tela inicial da aplicação para disponibilizar mais uma função, a função `Saldo` que fará a chamada a função `ImprimirSaldo` da classe
3. Altere a função `Extrato` para construir um extrato em string utilizando a classe `StringBuilder`, ao invés de utilizar a função `Console.WriteLine`. Utilize as funções `PadLeft` e `PadRight` para que o extrato seja exibido com os valores alinhados.
4. Na propriedade `CPF` da classe `conta`, altere o código para que no método de acesso `get` o `CPF` seja retornado com 11 dígitos, completando com zeros a esquerda
5. Utilizando a função `Split`, altere a propriedade `NomeCorrentista` para que ela armazene o nome e sobrenome em variáveis separadas.

### **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 capítulo 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 cópia, 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 instancia 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 importante 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 instancia 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:\\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:\\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 sequencia 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 estes 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 exemplo acima especificamos opções de compartilhamento de arquivos. Isto ocorre pois 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("TESTE1");  
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 nova 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");
```

## ***Exercício Prático 1 – Lendo arquivos texto***

1. Abra o projeto criado no exercício anterior
2. No início da função Main, faça a leitura do arquivo Contas.TXT. O arquivo é um arquivo com informações separadas por vírgula, que contém o seguinte

Agencia, NumeroDaConta, NomeCorrentista, TipoPessoa, Isento, Saldo

3. Utilizando o comando Split, separe os valores recebidos no arquivo em um array de strings
4. Crie um novo ArrayList para armazenar instancias da classe ContaBancaria baseada nas contas lidas no arquivo texto
5. Altere a função principal para que ao informar o número da conta, na entrada do programa, o programa tente efetuar a busca da conta especificada no array com as informações do arquivo. Caso a conta seja encontrada, o menu de opções deve ser exibido. Caso a conta não seja encontrada, deverá ser exibida a mensagem “Conta Inválida” e uma nova conta deverá ser solicitada.

## ***Exercício Prático 2 – Gravando arquivos texto***

1. Abra o projeto criando no capítulo anterior
2. Implemente a função de GravaLog para que as logs sejam armazenadas em um arquivo texto, contendo a descrição e a data e hora de gravação da log
3. Coloque algumas logs durante o processo de saque e depósito da classe conta
4. Teste a aplicação
5. Altere a aplicação de console para que ao iniciar o programa sejam apresentadas duas opções ao usuário
  - a. Abrir Conta
  - b. Criar Nova Conta
  - c. Sair
6. Caso o usuário solicite a primeira opção, ele deverá informar a conta a partir das contas armazenadas no arquivo, conforme já fizemos no exercício anterior
7. Caso o usuário solicite abrir uma conta, deverão ser solicitados os dados para abertura da conta. A partir destes dados, deverá ser gerada uma nova linha no arquivo Contas.TXT, contendo os dados da conta

## ***Exercício Prático 3 – Alterando o arquivo de contas***

1. Altere o programa para que ao solicitar a opção Sair, o programa gere um novo arquivo Contas.TXT, gravando as informações de saldo alteradas no arquivo para todas as contas do array de contas.

## **4. Orientação à 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 à 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)
    {
        //...
    }
}
```

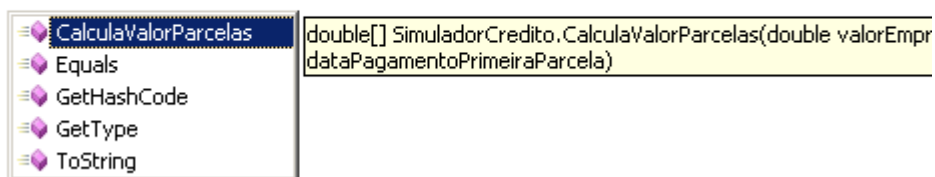
```
}  
}
```

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 utilizado 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();
```



### ***Exercício 1 – Implementando Herança***

1. Implemente uma nova classe com o nome de ContaEspecial que herde a classe conta corrente e que tenha as seguintes características adicionais
  - a. Uma variável membro Double com o nome de m\_LimiteCredito
  - b. Uma propriedade LimiteCredito, que funcione somente para leitura
  - c. Uma variável membro m\_TaxaJuros do tipo Double
2. Remove o teste do limite de crédito da lógica da classe ContaBancaria (esta lógica será utilizada apenas na classe ContaEspecial0)
3. Implemente uma nova classe com o nome de ContaPoupanca que herde as características da conta corrente
4. Implemente a propriedade DiaAniversario na classe ContaPoupanca
5. Implemente o método AplicarReajuste que aceite um parâmetro Double com o nome de rendimento. Não é necessário implementar o método ainda.

### ***O modificador de acesso protected***

As classes que herdaram 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 estes métodos possuírem visibilidade private, 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 permitiremos 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;
    }
}

```

## ***Exercício Prático 2 – Implementando visibilidade protected***

1. Altere a declaração das variáveis membro da classe `ContaCorrente` para `protected`
2. Implemente na classe `ContaPoupanca` o método `AplicarReajuste` alterando a variável `m_Saldo` a partir do reajuste informado. Caso a data de aniversário da poupança não seja

igual a data de hoje, o reajuste não deverá ser aplicado

3. Altere o arquivo contendo as contas para incluir mais uma flag que contenha o tipo de conta a ser utilizado
4. Altere o código para que ao ler o arquivo a classe correta de conta seja criada e armazenada no array de contas
5. Crie uma propriedade adicional na classe conta bancária que devolva um Enum chamado TipoConta
6. Crie um enum chamado tipo conta, com os seguintes valores
  - a. ContaCorrente
  - b. ContaPoupanca
  - c. ContaEspecial
7. Altere os métodos que gravam o arquivo de contas para que ele contenha o tipo de contas utilizado

## ***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 comportamento de funções da classe pai, reimplementando métodos, podemos utilizar o modificador virtual, conforme o exemplo a seguir.

```
class ContaCorrente
{
    protected double m_Saldo;

    public virtual double Sacar(double valor)
    {
```

```
        if (valor < m_Saldo)
            return 0;

        m_Saldo -= valor;

        return valor;
    }
}
```

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;
    }
}
```

## 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 modificador 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 a conversão de dados de classes pais para classes filhas e como esta implementação leva ao conceito de polimorfismo.

Quando construímos uma instancia de uma classe especializada (ex. ContaEspecial) podemos atribuir esta instancia 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 = ctaA;
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 instancia continua sendo uma instancia de conta especial.

O poliformismo acontece justamente na chamada do método Sacar. Mesmo estando armazenada em uma variável ContaCorrente, a instancia da classe ContaEspecial continua se comportando normalmente, ou seja, nas chamadas 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 explícita. Isto porque a variável conta corrente pode não conter uma instancia 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 = ctaA;
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 *System.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 instâncias de classe sendo convertidas para o tipo *object*.

```
Cliente cli = new Cliente();
CustomerBO 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 um 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 numero 10
// em um object
object obj = 10;

// unboxing da referencia
// 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;
    }
}
```

### ***Exercício Prático 3 – Implementando Métodos virtuais***

1. Marque os métodos sacar e depositar da classe ContaCorrente com o modificador virtual
2. Marque o método ValidarSaque como virtual
3. Reimplemente o método ValidarSaque na classe ContaEspecial de forma a considerar o limite de crédito
4. Reimplemente o método Sacar na classe ContaPoupanca para que não seja descontado o CPMF
5. Reimplemente a propriedade TipoConta em cada uma das classes de conta para que elas retornem o tipo correto de conta que deve ser utilizado
6. Reimplemente o método ToString da classe conta bancária para que este devolva as informações relativas a conta como Número da Conta, Nome do Correntista, etc.

### ***Classes parciais***

Outro recurso interessante disponível no C# é a possibilidade de implementarmos uma classe em diversos arquivos. Isto é possível utilizando o modificador parcial na declaração da classe. Todas as partes da classe devem conter este modificador. O compilador, no momento da compilação, junta

todas as definições da classe em um arquivo único e efetua a compilação. Um exemplo de uma classe distribuída utilizando o modificador partial pode ser visto a seguir.

#### Arquivo1.cs

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

#### Arquivo2.cs

```
partial class ContaPoupanca
{
    DateTime m_DataAniversario;

    public DateTime DataAniversario
    {
        get
        {
            return m_DataAniversario;
        }
    }
}
```

## 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 Prático 4 – Implementando documentação XML***

1. Implemente a documentação XML para os métodos da classe ContaCorrente, ContaPoupanca e ContaEspecial
2. Implemente a documentação XML para as classes ContaCorrente, ContaPoupanca e ContaEspecial
3. Verifique se os comentários inseridos na documentação aparecem no intellisense



## **5. Interfaces e coleções**

## Definindo interfaces no C#

O conceito de interfaces é um conceito avançado de implementação de sistemas orientados a objetos. Apesar de aparentemente parecer complicado, o conceito de interface é extremamente simples e permite a construção de sistemas altamente flexíveis.

A interface funciona como uma espécie de protocolo utilizado para comunicação entre objetos. Para tentarmos ilustrar o funcionamento de interfaces, vamos utilizar o exemplo a seguir.



No exemplo acima temos uma classe chamada Impressora que faz interface com uma impressora capaz de imprimir documentos de qualquer espécie. Seu sistema trabalhar com uma série de documentos. A classe Impressora tem um método chamado imprimir, que deve receber como parâmetro um documento para ser impresso. A grande dúvida é, que tipo devemos utilizar para este parâmetro?

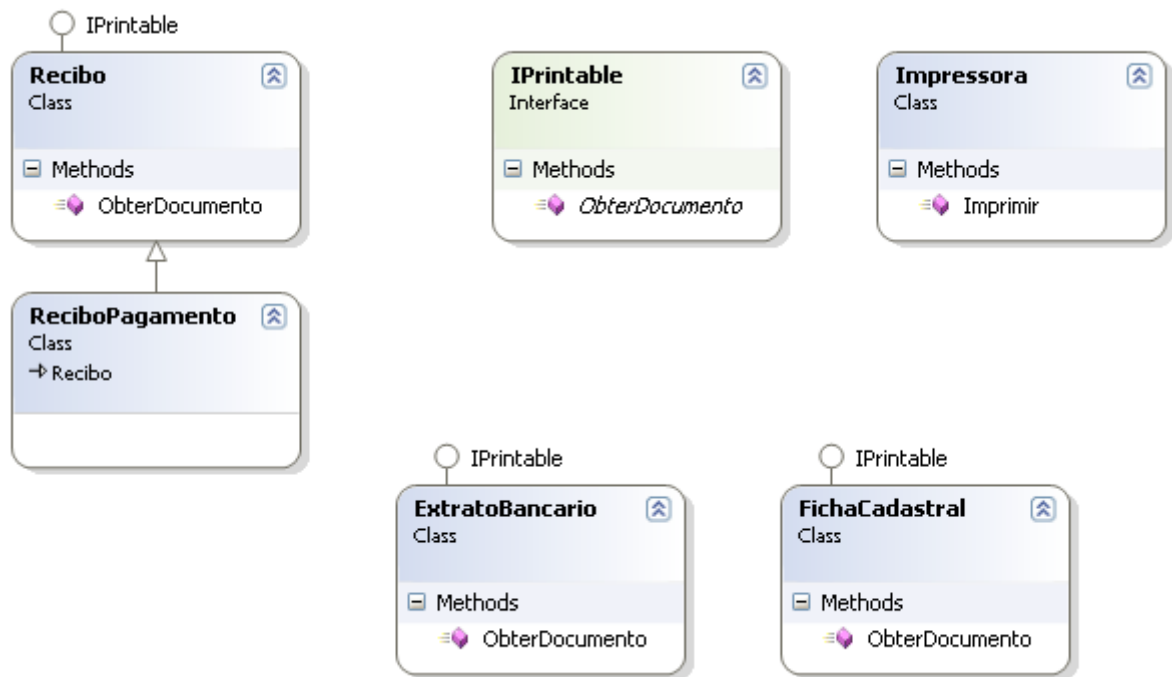
Caso quiséssemos imprimir recibos, poderíamos fazer com que a classe impressora recebesse uma instância da classe pai Recibo. O problema é que isto não permitiria a impressão de documentos da classe ExtratoBancario ou mesmo da classe FichaCadastral.

Outra opção seria criar diversos overloads do método imprimir, permitindo a chamada com diversos tipos de parâmetro. O problema deste tipo de abordagem é que a cada documento novo que surgisse, seria necessário implementar alterações na classe Impressora, para criar um novo overload e um novo documento.

As interfaces servem para resolver justamente situações como esta. As interfaces funcionam como uma espécie de protocolo, que permite objetos distintos (e que muitas vezes herdam de classes diferentes) possam ser utilizados como parâmetro ou mesmo como tipo de retorno para outros objetos.

No exemplo acima, poderíamos implementar uma interface com o nome IPrintable, que obrigaria cada uma das classes que implementa tal interface a disponibilização de um método

ObterDocumento, para obter as informações do documento. Desta forma, a classe impressora poderia receber qualquer objeto como parâmetro para o método imprimir, desde que este parâmetro implementasse a interface IPrintable.



## Implementando interfaces no C#

A criação de interfaces no C# é semelhante a criação classes, com a diferença que utilizamos a palavra reservada interface ao invés da palavra class, conforme o exemplo a seguir.

```
public interface IPrintable
{
    void ObterDocumento();
    int NumeroPaginas();
}
```

No que a declaração dos métodos nas interfaces não possuem modificador de acesso (public, private, protected). Isto ocorre pois todos os membros declarados na interface devem ser obrigatoriamente públicos.

Outro detalhe importante que deve ser observado é que os métodos não possuem implementação nas interfaces. Conforme já comentamos, a interface funciona apenas como um protocolo, ou seja, não existe qualquer implementação dentro da interface.

Para fazermos com que uma classe implemente uma interface, devemos utilizar a mesma notação da herança de classes, ou seja, colocar após a declaração da classe o símbolo “:” e o nome da interface. Caso a classe já herde as características de uma outra classe ou mesmo de outra interface, podemos informar uma interface adicional separando cada uma delas por vírgula.

```
public class Recibo : IPrintable
{
    public void ObterDocumento()
    {

    }
}
```

## ***Diferença entre interfaces e classes abstratas***

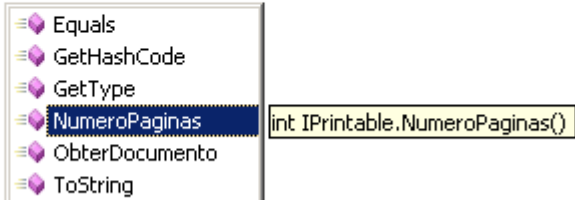
É importante salientar a diferença entre as classes abstratas e as interfaces. As classes abstratas possuem todas as funcionalidades de uma classe normal, ou seja, podem ter campos, implementação em métodos, campos privados, etc. Já as interfaces não podem conter estas estruturas, pois elas definem apenas um padrão de assinatura o qual um determinado objeto deve implementar.

## ***Utilizando interfaces como parâmetros ou retorno de funções***

A grande vantagem na utilização de interfaces é a possibilidade de utilizarmos estas estruturas como parâmetros de métodos ou mesmo como valores de retorno de uma função.

No exemplo que avaliamos acima, poderíamos implementar o método Imprimir da classe impressora recebendo um parâmetro do tipo IPrintable, conforme o exemplo a seguir.

```
public class Impressora
{
    public void Imprimir(IPrintable documento)
    {
        documento.
    }
}
```



Apesar de o método imprimir poder receber objetos dos mais variados tipos como parâmetro através da classe IPrintable, note que as únicas funções disponíveis para chamada são as funções implementadas na interface.

É importante definirmos uma interface com todos os métodos necessários para implementarmos a funcionalidade desejada.

## ***As interfaces padrão do .NET Framework***

O conceito de interface é amplamente utilizado na biblioteca de classes do .NET Framework. Você pode perceber que muitos métodos que chamamos até aqui aceitam como parâmetro

interfaces, ao invés de classes. Como exemplo podemos pegar a classe `ArrayList`, que já utilizamos em capítulos anteriores. Podemos verificar que o método `AddRange` recebe como parâmetro qualquer objeto que implemente uma interface `ICollection`, o que abrange arrays de qualquer natureza, `HashTables`, `SortedList`s, etc.

```
public void Imprimir(IPrintable documento)
{
    ArrayList lista = new ArrayList();

    lista.AddRange(|
void ArrayList.AddRange (ICollection c)
c: The System.Collections.ICollection whose elements should be added to the end of the System.
cannot be null, but it can contain elements that are null.
```

### ***A interface IEnumerable***

A interface `IEnumerable` é uma interface padrão do .NET framework que permite a implementação de enumeradores. Sempre que queremos permitir que o objeto possa ser utilizado em um comando `foreach`, devemos implementar a interface `IEnumerable`. Veja o exemplo a seguir.

```
public class Usuarios : IEnumerable
{
    private string[] nomesArray;
    public Usuarios()
    {
        nomesArray = new string[4];
        nomesArray[0] = "Nome 1";
        nomesArray[1] = "Nome 1";
        nomesArray[2] = "Nome 1";
        nomesArray[3] = "Nome 1";
    }
    public IEnumerator GetEnumerator()
    {
        return nomesArray.GetEnumerator();
    }
}
```

### ***A interface ICloneable***

A interface `ICloneable` obriga o objeto a implementar o método `Clone` que deve criar uma cópia exata do objeto. Este tipo de interface é útil para casos em que queremos garantir que o objeto possa ser duplicado durante a execução do programa. Veja o exemplo a seguir.

```
public class Cliente : ICloneable
{
    string m_Nome;

    public Cliente(string nome)
    {
        m_Nome = nome;
    }

    public object Clone()
    {
        return new Cliente(m_Nome);
    }
}
```

### ***A interface IComparable***

A interface IComparable é utilizada para permitir que objetos de uma classe possam ser comparados com objetos de outra classe. A interface IComparable é útil em situações em que temos que implementar ordenação em coleções de itens de nossa classe. Veja o exemplo.

```
public class Cliente : IComparable
{
    long m_CPF;

    public Cliente(long cpf)
    {
        m_CPF = cpf;
    }

    public int CompareTo(object obj)
    {
        Cliente cli = (Cliente)obj;

        if (cli.m_CPF > this.m_CPF)
            return 1;
        else if (cli.m_CPF < this.m_CPF)
            return -1;
        else
            return 0;
    }
}
```

Na implementação da interface `Comparable` devemos retornar o valor 1 caso o valor da classe seja maior que o valor da classe comparada. Caso o valor seja menor, deverá ser retornado o valor -1. Caso as duas classes tenham valores iguais, deverá ser retornado o valor 0.

Se tivéssemos por exemplo criado um array de objetos `Cliente`, poderíamos perfeitamente utilizar a função `Array.Sort` para ordenar os elementos `Cliente` do Array, tendo em vista que implementamos a interface `Comparable` na classe.

### ***Exercício Prático 1 – Implementando Interfaces***

1. Abra o projeto que contém as classes de contas correntes
2. Crie uma interface chamada `IConta` que contenha declaração dos métodos `Sacar`, `Depositar` e da propriedade `Saldo`
3. Encapsule a lista de contas utilizada no exercício anterior em uma classe chamada `ContasBanco`
4. Implemente a leitura do arquivo de contas em um método desta nova classe
5. Implemente a gravação do arquivo de contas em um novo método desta nova classe
6. Implemente a interface `IEnumerable` na classe `ContasBanco` para que seja possível enumerar as contas do banco
7. Implemente a interface `Comparable` na classe conta bancária para que sejam possível a comparação entre contas utilizando o número da conta
8. Implemente a classe `ICloneable` na classe conta para que seja possível a criação de uma nova conta a partir de uma conta existente.

## **6. 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 instancias de classes. Conforme vimos, as classes são apenas especificações de uma determinada entidade do sistema que determinam o seu comportamento. Para criarmos uma instancia de uma classe (um objeto) devemos criar uma variável do tipo desta classe e utilizar o comando new para alocar uma nova instancia da classe. Podemos alocar quantas instancias 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 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 instancia de qualquer objeto, o framework guarda uma referencia desta instancia. No momento em que as variáveis que referenciam esta instancia saem de escopo e nenhuma variável aponta para aquela referencia de objeto, o objeto é eliminado automaticamente.

## ***O Garbage Collector***

O processo responsável por eliminar referencias 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 collector 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. Estas funções são chamadas funções destrutoras. Em geral utilizamos estas funções para destruímos 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 a 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 S01VWClass objetoNaoGerenciavel;

    public Funcionario()
    {
        objetoNaoGerenciavel = new
            S01VWClass();
    }

    ~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 destrutor para a classe, de forma a liberar 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 S01VWClass objetoNaoGerenciavel;

    public Funcionario()
    {
        objetoNaoGerenciavel = new
            S01VWClass();
    }

    ~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 vez. 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 Prático 1 – Implementando a interface IDisposable***

1. Abra o projeto de criado nos exercícios anteriores
2. Implemente a interface IDisposable em todas as classes de contas correntes
3. Implemente um finalizador nas classes. Por enquanto não é necessário implementar os métodos, faremos isto posteriormente.
4. Altere a classe de gravação de log para que o arquivo de log seja aberto no construtor da classe e seja fechado pelo destrutor do objeto ou pelo método Dispose
5. Altere a classe de leitura de contas para abrir o arquivo de contas no seu construtor e eliminar esta referencia em seu destrutor ou no método Dispose

## **7. 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 avali

```
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.

Neste trecho de código podem 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.

### ***Exercício Prático 1 – Utilizando tratamento estrutura de exceções***

1. Abra o projeto do exercício criado no módulo anterior
2. Insira um tratamento de exceção para a leitura do arquivo de contas. Caso o arquivo de contas não seja encontrado ou mesmo não tenha permissão, o erro deverá ser tratado e nenhuma conta deverá ser lida
3. Faça um tratamento na inclusão de contas no arraylist de contas para que caso a linha esteja mal formatada (não tenho o número de campos suficientes) a exceção seja tratada e o registro do arquivo seja ignorado. Nestes casos deverá ser gerada uma log.
4. Altere a rotina de tratamento de log para que ela não gere qualquer tipo de erro caso não consiga gravar um registro de log
5. Faça o teste do tratamento de erros trocando o arquivo Contas.Txt pelo arquivo ContasErro.Txt
6. Coloque um tratamento de erro na função Main para que, caso algum erro grave aconteça na aplicação, este seja logado e a aplicação retorne para o primeiro menu (Abrir conta ou Nova Conta).

### ***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 Prático 2 – Utilizando exceções customizadas***

1. Crie um novo tipo de exceção chamado de Exception que contenha um campo adicional chamado RetCode
2. Faça com que o sistema lance uma exceção nos casos em que for atribuído um nome em branco para o nome do correntista
3. Faça com que o sistema lance uma exceção nos casos em que um valor inválido para o tipo de conta for atribuído
4. Faça com que o sistema lance uma exceção caso o usuário informe uma opção de menu inválida. Trate este erro fazendo com que o sistema volte para a primeira opção do menu.

## **8. Delegates e eventos**

## Introdução

A maioria de nós já foi exposto 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 conhecido 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 construímos 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;

    // Constructor
    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.

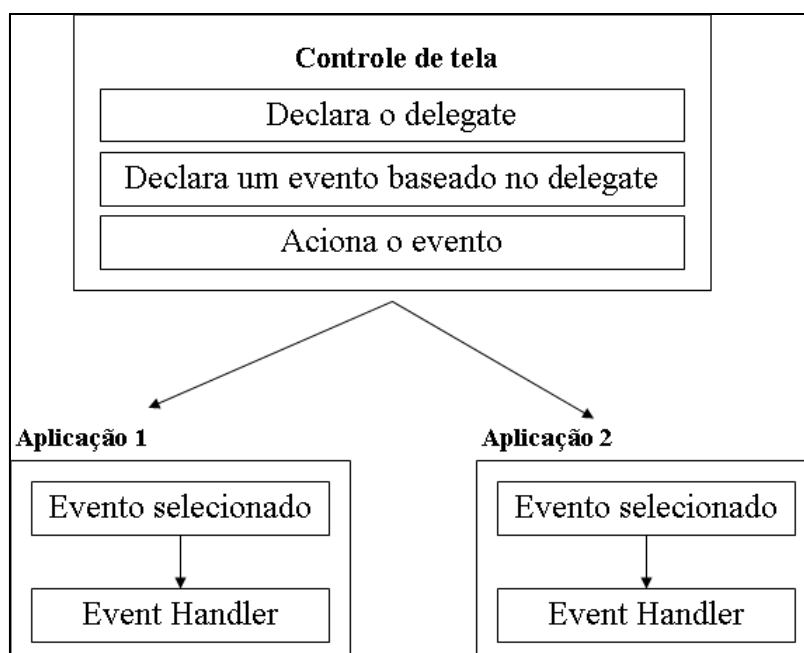
## Exercício Prático 1 – Utilizando delegates

1. Abra a solução utilizada nos exercícios anteriores
2. Crie um novo delegate na classe ContaBancaria com o nome de GravacaoLog recebendo um parâmetro string.
3. Declare uma variável pública do tipo do delegate criado no passo anterior
4. Nos locais onde antes era chamada a função de gravação da classe de log, altere a função para chamar uma instancia do delegate
5. Associe a variável pública da instancia da classe contaBancaria com a função gravalog da classe de log e verifique se as logs continuam sendo gravadas
6. Cria uma nova log para mostrar os eventos na tela (Console.WriteLine) ao invés de salvar em arquivos.
7. Associe a função de gravação de log desta classe ao delegate e verifique o resultado

## Eventos

O modelo de eventos do C# é muito semelhante ao modelo de outras linguagens de programação, inclusive ao do visual basic. Neste modelo trabalhamos com o conceito de publicador e subscritor, onde o publicador irá enviar somente os eventos para os subscritores que subscreveram aos seus eventos.

No C#, uma classe pode publicar uma série de eventos para que outras classes possam subscrever. Quando a classe publica um evento, todas as classes que subscreveram a este evento são notificadas, conforme o exemplo a seguir.



## Convenções utilizadas na criação de eventos

Para criarmos eventos, devemos utilizar as seguintes convenções:

- Eventhandlers no .NET framework deve retornar um valor void e receber dois parâmetros
- O primeiro parâmetro representa a origem do evento ( o Publisher )
- O segundo parâmetro é um objeto derivado da classe EventArgs
- Eventos são propriedades da classe que está publicando eventos
- A palavra “event” define como os subscritores irão acessar o evento

Vamos modificar o exemplo que utilizamos nos delegates para utilizarmos uma implementação através de eventos.

```
/* ===== Publicador ===== */  
public class MinhaClasse  
{  
    public delegate void LogHandler(string message);  
  
    // Declaração do evento  
    public event LogHandler Log;  
  
    public void Process()  
    {  
        OnLog("Process() begin");  
        OnLog("Process() end");  
    }  
  
    protected void OnLog(string message)  
    {  
        if (Log != null)  
        {  
            Log(message);  
        }  
    }  
}  
  
public class FileLogger  
{  
    FileStream fileStream;
```

```

StreamWriter streamWriter;

// Constructor
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();
}
}

/* ===== Subscritor ===== */
public class TestApplication
{
    static void Logger(string s)
    {
        Console.WriteLine(s);
    }

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

        // Subscreve o evento para duas funções
        minhaClasse.Log += new minhaClasse.LogHandler(Logger);
        minhaClasse.Log += new minhaClasse.LogHandler(fl.Logger);

        minhaClasse.Process();

        fl.Close();
    }
}

```

No exemplo acima temos a classe MinhaClasse declarando um evento Log. A chamada ao evento Log é semelhante a chamada da delegate que utilizamos anteriormente.

Na classe subscritora, temos a função Main subscrevendo o evento Log para as duas funções de log, uma de arquivo e outra estática.

## ***Exercício Prático 2 – Utilizando eventos***

1. Abra a solução utilizada nos exercícios anteriores
2. Crie dois eventos na classe ContaBancaria, chamados OnSaque e OnDeposito.
3. Faça a chamada destes eventos dentro das funções de Saque e Deposito
4. Altere o código da função Main para mapear os eventos de saque e depósito, gravando uma log para cada um deles

## **9. Técnicas avançadas de construção de tipos**

## Construindo tipos com indexadores

Os indexadores permitem a utilização de índices, da mesma forma que os arrays, para obtermos valores armazenados neste objeto.

A sintaxe de um indexadores é muito semelhante a sintaxe de uma propriedade, com a diferença que devemos utilizar o nome `this` e informar um parâmetro que representará o índice.

```
public dataType this[int index]
{
    get
    {
        return aValue;
    }
    set
    {
        // ...
    }
}
```

A criação do indexadores permite a utilização da sintaxe utilizando os colchetes [ ] onde a função `set` será chamada para setar os valores e a função `get` será utilizada para obtermos os valores. O tipo do índice a ser utilizado deve ser indicado na declaração do indexador. O exemplo a seguir mostra a declaração e o uso de indexadores em um tipo chamado `DadosCliente`.

```
using System;
using System.Xml;

public class DadosCliente
{
    XmlDocument documento;

    public string this [string campo]
    {
        get
        {
            return documento.SelectSingleNode("/xml/dados/registrouuario/" +
campo).InnerText;
        }
        set
        {
            documento.SelectSingleNode("/xml/dados/registrouuario/" + campo).InnerText =
value;
        }
    }
}
```

```

public DadosCliente(string sXml)
{
    documento = new XmlDocument();
    documento.LoadXml(sXml);
}
}

public class Programa
{
    public void Main()
    {
        string sXml = "<xml><dados><registrouuario    cpf=\"12345678901\"
nome=\"teste\"/></registrouuario></dados></xml>";

        DadosCliente cli = new DadosCliente(sXml);

        string cpf = cli["cpf"];
        string nome = cli["nome"];
    }
}

```

No exemplo acima estamos utilizando uma classe e um indexador para abstrair a complexidade de pesquisa em um arquivo xml. Utilizamos o indexador para buscar os campos apenas pelo nome, ao invés de utilizarmos as classes de manipulação de xml.

## ***Overload de operadores***

Outro recurso avançado que temos no C# é a possibilidade da criação de overload de operadores. Através deste recurso podemos fazer com que as classes criadas possam ser utilizadas em operações que envolvam os operadores de soma, subtração e etc.

A definição de um overload de operadores é muito semelhante a definição de um método, com a diferença que utilizamos a palavra reservada `operator` e o símbolo do operador ao invés do nome do método. O overload de operadores deverá ter um tipo de retorno que será utilizado na operação bem como os parâmetros, que representam o funcionamento da operação.

O exemplo a seguir mostra a utilização de overload de operadores em uma classe chamada `Titulo` que representa um título bancário.

```

public class Titulo
{
    public double valor;
    public string codigoCedente;

    //...
}

```



```

public string CodigoBarras
{
    get { }
    set { }
}

public static Titulo operator +(Titulo t1, Titulo t2)
{
    Titulo tret = new Titulo();
    tret.codigoCedente = t1.codigoCedente;
    tret.valor = t1.valor + t2.valor;

    return tret;
}

public class Program
{
    public void Main()
    {
        Titulo t1 = new Titulo();
        t1.valor = 15.50;

        Titulo t2 = new Titulo();
        t2.valor = 58.90;

        // Overload do operador +
        Titulo t3 = t1 + t2;

    }
}

```

No exemplo acima criamos um overload para o operador + que permite a soma de duas classes do tipo Titulo. Através da operação de soma, somamos os valores dois titulos gerando um novo titulo que é devolvido como resultado.

Note que a função de overload do operador é marcada com o atributo static. Este tipo de construção é obrigatória para overload de operadores.

## ***Rotinas de conversão***

Da mesma forma que criamos os overloads de operadores, podemos criar overloads capazes de realizar conversões de dados. Podemos por exemplo fazer com que uma classe do tipo Titulo possa ser convertida para um valor Double, onde será atribuído o valor total do título.

```

public class Titulo
{
    public double valor;

    //...

    public static explicit operator double(Titulo t1)
    {
        return t1.valor;
    }
}

public class Program
{
    public void Main()
    {
        Titulo t1 = new Titulo();
        t1.valor = 15.50;

        double valor = (double) t1;
    }
}

```

No exemplo acima criamos uma função de conversão do tipo Titulo para o tipo double. A declaração da função customizada de conversão é semelhante ao overload de operador, porém utilizada o tipo de destino ao invés do operador utilizado. Note também que o operador de conversão possui um parâmetro explicit, que indica que a conversão para este tipo de dado deve ser realizada de forma explícita (utilizando um operador de cast). Poderíamos ter alterado esta conversão para que ela fosse feita de forma implícita, sem a necessidade de um operador de cast, conforme exemplo a seguir.

```

public class Titulo
{
    public double valor;

    //...

    public static implicit operator double(Titulo t1)
    {
        return t1.valor;
    }
}

public class Program

```

```
{  
    public void Main()  
    {  
        Titulo t1 = new Titulo();  
        t1.valor = 15.50;  
  
        double valor = t1;  
    }  
}
```

## ***Exercício Prático 1 – Utilizando indexadores e overload de operadores***

1. Abra o exercício criado no módulo anterior
2. Altere a classe que contém a relação de contas para que ela suporte também um indexador e devolva um tipo ContaBancaria como resultado
3. Crie um overload para o operador + para a classe OperacaoContabil, somando o valor das duas operações e concatenando o valor dos históricos. No campo data coloque a data atual (Now)
4. Crie um overload para o operador -, de forma semelhante ao operador + da classe OperacaoContabil
5. Crie um método chamado SomaOperacoes que some todos os elementos da classe OperacaoContabil presentes em uma classe conta
6. Crie um novo item de menu chamado “Total de Operações” que exiba a soma de todos os itens SomaOperacoes contidos na lista de uma conta específica.
7. Execute o programa e teste as novas funcionalidades

## ***Comandos avançados no C#***

Além dos recursos de indexadores, overload de operadores e de funções de conversão, o C# ainda suporta alguns comandos avançados que devem ser observados.

### **Os comandos checked e unchecked**

Como você deve saber, em qualquer linguagem de programação os valores numéricos possuem um limite máximo de capacidade. Um valor inteiro por exemplo, pode armazenar um número de até 2.147.483.647, um byte é capaz de armazenar um valor de até 255. O que talvez você não tenha percebido até o momento é que o C# não faz qualquer tipo de validação destes limites quando estamos executando operações com números. Veja o exemplo a seguir.

```
byte n1 = 100;
```

```
byte n2 = 200;

// Não resulta em erro
// mas sim em um número, 44
byte n3 = (byte)(n1 + n2);
```

O C# trabalha com o modelo padrão chamado unchecked. No modelo unchecked o programa não faz qualquer tipo de verificação de limites nos números, o que permite um incremento significativo de performance, mas pode acarretar em problemas de lógica. O exemplo acima é equivalente ao exemplo mostrado a seguir, tendo em vista que o modelo unchecked é o default do C#.

```
unchecked
{
    byte n1 = 100;
    byte n2 = 200;
    // Não resulta em erro
    // mas sim em um número, 44
    byte n3 = (byte)(n1 + n2);
}
```

Se quiséssemos que o C# lançasse uma exceção no caso de um estouro em algum número, poderíamos envolver a operação de soma em um bloco checked. Este bloco garante que em qualquer operação numérica os limites das variáveis serão testados.

```
checked
{
    byte n1 = 100;
    byte n2 = 200;

    // Exception aqui !
    byte n3 = (byte)(n1 + n2);
}
```

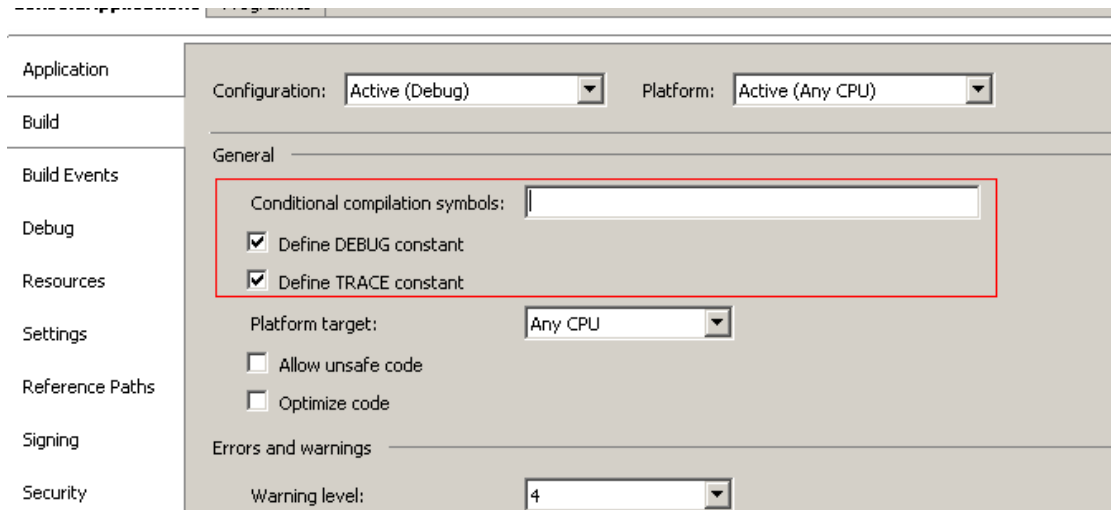
## ***Exercício Prático 2 – Utilizando a diretiva checked***

1. Abra o exercício criado no módulo anterior
2. Nas operações de soma, como por exemplo a operação de soma de registros contábeis, altere o código para que as operações sejam feitas de forma verificada, utilizando o operador “checked”

## Diretivas de processamento

As diretivas de processamento são comandos úteis no C# para os casos em que queremos que um determinado bloco de código seja compilado somente em algumas situações.

A compilação condicional é realizada através de variáveis de compilação, que são definidas nas propriedades de projeto do visual studio.



Podemos testar se uma determinada variável está definida ou não e dependendo desta informação podemos compilar um bloco de código ou não.

Para utilizarmos este recurso, devemos utilizar os comando #if, #else e #endif, conforme exemplo a seguir.

```
class Program
{
    static void Main(string[] args)
    {
        #if DEBUG
            Console.WriteLine("Aplicação em teste!");
        #else
            Console.WriteLine("Aplicação em produção");
        #endif
    }
}
```

Outro recurso interessante é a possibilidade de criação de regiões de código. Podemos utilizar estas regiões para organizar blocos de funções afim de tornar o código mais legível, conforme o exemplo a seguir.

Utilizando regiões, tornamos o código mais legível pois o visual studio nos permite esconder o código em uma determinada região.

```
partial class Teste
```

```

{
    #region Propriedades

    private int m_prop1;

    public int Propriedade1
    {
        get { return m_prop1; }
        set { m_prop1 = value; }
    }

    private int m_prop2;

    public int Propriedade2
    {
        get { return m_prop2; }
        set { m_prop2 = value; }
    }

    #endregion
}

```

Na ilustração a seguir a região contendo as propriedades está fechada.

```

partial class Teste
{
    Propriedades
}

```

## O atributo [Conditional]

Outra opção importante disponível do C# é o atributo Conditional. Este atributo é semelhante a compilação condicional, mas pode ser aplicado diretamente em métodos. A vantagem deste atributo é que além de remover a declaração do método ele remove também as chamadas deste método presentes em todo o código. Confira o exemplo a seguir.

```

class Program
{
    [System.Diagnostics.Conditional("DEBUG")]
    static void GravaLog()
    {
        // ... grava log
    }
}

```

```
static void Main(string[] args)
{

    Console.WriteLine("Aplicação em teste!");
    GravaLog("TESTE");

}
}
```

No exemplo acima, a função GravaLog bem como suas chamadas só será compilada no caso de o símbolo DEBUG estar definido (ou seja, na compilação em modo debug). Caso este símbolo não esteja definido, o método e suas não serão compilados.

### ***Exercício Prático 3 – Compilação condicional***

1. Abra o exercício criado no módulo anterior
2. Altere a função de gravação de log para que ela só seja compilada caso a compilação esteja sendo feita em modo DEBUG
3. Execute a aplicação e verifique o resultado

## **10. 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 frequência, 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 numero 10
// em um object
lista.Add(10);

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

## ***O namespace System.Collections.Generic***

A biblioteca System.Collections.Generic 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.Generic, podemos enumerar as seguintes.

- ICollection<T>
- IComparer<T>
- IDictionary<K, V>
- IEnumerable<T>
- IEnumerator<T>
- IList<T>

## Analizando 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 Prático 1 – Utilizando coleções genéricas***

1. Abra o exercício utilizado nos exercícios anteriores
2. Altere a coleção que armazena as operações contábeis para que esta passe a ser uma coleção genérica ao invés de um ArrayList
3. Altere a coleção que armazena as contas do arquivo Contas.TXT para que ela seja uma coleção genérica do tipo ContaBancaria
4. Execute a aplicação e verifique o resultado

## ***Criando classes e estruturas genéricas***

Da mesma forma que criamos métodos genéricos, podemos declarar classes ou mesmo estruturas genéricas. O exemplo a seguir mostra uma estrutura genérica, capaz de aceitar qualquer tipo de dado.

```
public class Coordenada<T>
{
    T m_v1;
    T m_v2;

    public Coordenada(T v1, T v2)
    {
        m_v1 = v1;
        m_v2 = v2;
    }

    public T X
    {
        get {return m_v1;}
        set { m_v1 = value; }
    }

    public T Y
    {
        get {return m_v2;}
        set { m_v2 = value; }
    }
}
```

No exemplo acima temos o exemplo de uma classe com o nome de coordenada, que pode receber uma coordenada de qualquer tipo. Podemos utilizar uma coordenada que aceite tanto inteiros como também strings.

```
public void Main()
{
    Coordenada<int> localizacao = new Coordenada<int>(10, 20);

    Coordenada<string> xadrez = new
        Coordenada<string>("A", "1");
}
```

No construtor de uma classe genérica, podemos utilizar a palavra reservada default para

inicializar as variáveis com os tipos não definidos para seus valores default. O C# inicializa valores numéricos com zero e valores reference type com null.

```
public class Coordenada<T>
{
    T m_v1;
    T m_v2;

    public Coordenada()
    {
        m_v1 = default(T);
        m_v2 = default(T);
    }

    public Coordenada(T v1, T v2)
    {
        m_v1 = v1;
        m_v2 = v2;
    }

    public T X
    {
        get {return m_v1;}
        set { m_v1 = value; }
    }

    public T Y
    {
        get {return m_v2;}
        set { m_v2 = value; }
    }
}
```

Podemos ainda criar um tipo que utilize mais de um tipo genérico. Para isto, basta informar após o primeiro tipo genérico uma vírgula e outra letra para representar o segundo tipo genérico, conforme o exemplo.

```
public class Coordenada<T, U>
{
    T m_v1;
    U m_v2;

    public Coordenada()
    {
        m_v1 = default(T);
    }
}
```

```

        m_v2 = default(U);
    }

    public Coordenada(T v1, U v2)
    {
        m_v1 = v1;
        m_v2 = v2;
    }

    public T X
    {
        get {return m_v1;}
        set { m_v1 = value; }
    }
    public U Y
    {
        get {return m_v2;}
        set { m_v2 = value; }
    }
}

```

No exemplo acima, utilizaríamos a classe com a seguinte notação.

```

Coordenada<int, int> localizacao = new
    Coordenada<int, int>(10, 20);

Coordenada<string, int> xadrez = new
    Coordenada<string, int>("A", 1);

```

## ***Criando interfaces genéricas***

Da mesma forma que as classes, podemos utilizar o recurso dos tipos genéricos na declaração de interfaces. A sistemática é exatamente a mesma utilizada em classes, conforme o exemplo a seguir.

```

public interface IBinaryOperations<T>
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
}

```

Para implementarmos uma interface com um tipo genérico, basta informarmos o tipo desejado na cláusula que associa a interface a classe, conforme o exemplo a seguir.

```
public class BasicMath : IBinaryOperations<double>
{
    public double Add(double arg1, double arg2)
    { return arg1 + arg2; }
    ...
}
```

### ***Exercício Prático 2 – Implementando classes genéricas***

1. Abra o projeto utilizado nos exercícios anteriores
2. Altere a classe OperacaoContabil para que os valores armazenados na classe utilizem tipos definidos dinamicamente
3. Altere os locais onde a classe é utilizada para que ela seja criada informando os tipos genéricos
4. Execute a aplicação novamente e verifique o resultado

## **11. .NET Assemblys**



## ***Introdução***

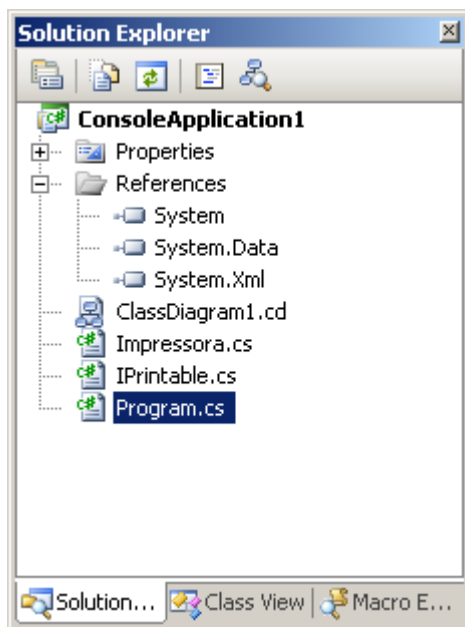
As aplicações que criamos até o momento geravam apenas um módulo, que quando executado produzia um resultado qualquer. Na prática, quando estamos trabalhando com um projeto .NET, a tendência é criarmos uma solução de forma mais modularizada, não colocando os componentes em um único projeto, mas sim em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

Para criarmos estes projetos de forma distribuída, devemos fazer uso dos assemblies. Os assemblies 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 assemblies***

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 referencia a outros assemblies.

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

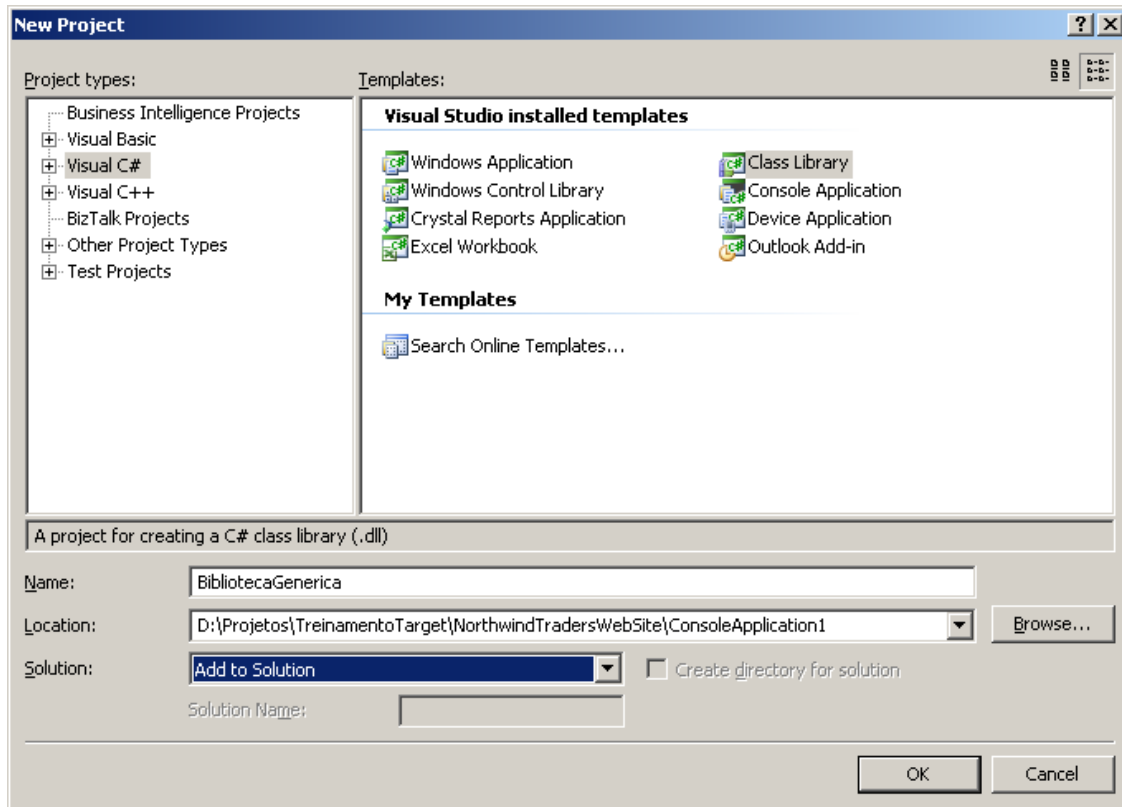


## ***Criando um assembly***

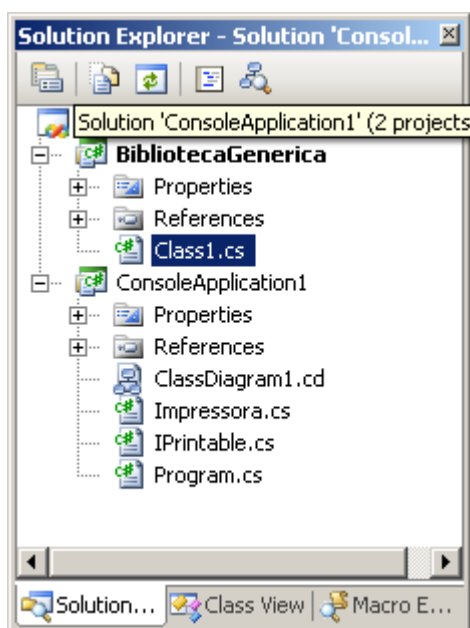
Os assemblies são representados por projetos do tipo Class Library. Estes projetos permitem a criação de um conjunto de classes em um módulo a parte, que terá como resultado final um arquivo com a extensão.dll (um assembly). Este assembly poderá ser posteriormente utilizado por outras aplicações.

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

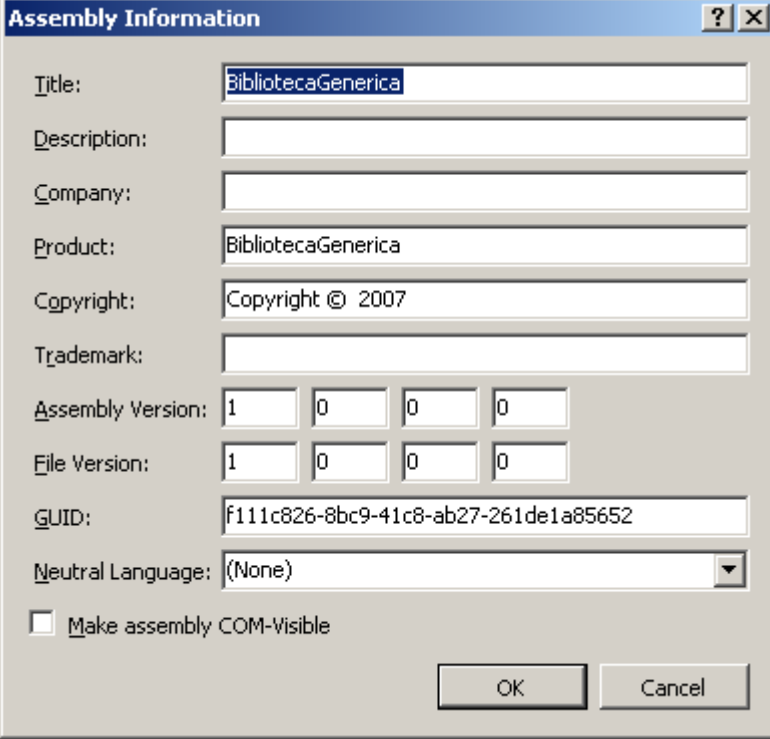


4. Informe o nome da biblioteca e clique em ok. Caso você já tenha um projeto aberto, poderá fazer com que este componente seja adicionado a mesma solução, desta forma você poderá trabalhar com vários projetos de diferentes componentes ao mesmo tempo.



## ***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”.

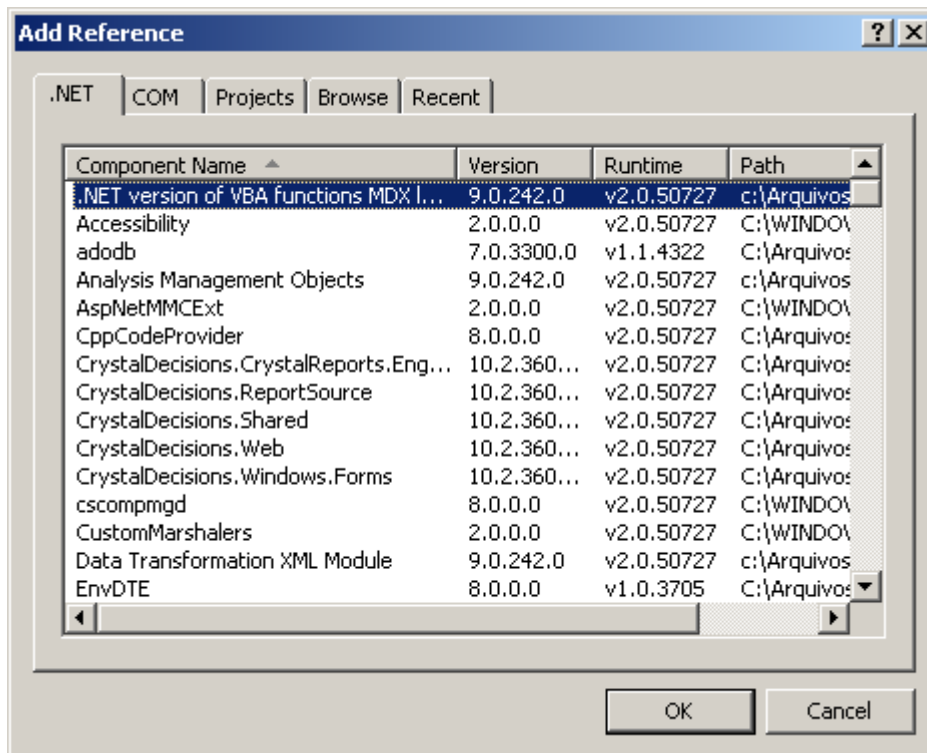


The image shows the 'Assembly Information' dialog box in a software development environment. The dialog has a title bar with a question mark and a close button. It contains several text input fields and a checkbox. The fields are labeled: Title, Description, Company, Product, Copyright, Trademark, Assembly Version, File Version, GUID, and Neutral Language. The 'Assembly Version' and 'File Version' fields are split into four boxes each, representing major, minor, build, and revision numbers. The 'GUID' field contains a long alphanumeric string. The 'Neutral Language' field is a dropdown menu currently set to '(None)'. At the bottom, there is a checkbox labeled 'Make assembly COM-Visible' and two buttons: 'OK' and 'Cancel'.

Title:	BibliotecaGenerica			
Description:				
Company:				
Product:	BibliotecaGenerica			
Copyright:	Copyright © 2007			
Trademark:				
Assembly Version:	1	0	0	0
File Version:	1	0	0	0
GUID:	f111c826-8bc9-41c8-ab27-261de1a85652			
Neutral Language:	(None)			
<input type="checkbox"/> Make assembly COM-Visible				
		OK Cancel		

## ***Referenciando um assembly***

Após criarmos o assembly, podemos referenciá-lo em outros projetos afim 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 assemblies (o GAC) através da aba .NET.

### ***Exercício Prático 1 – Criando componentes***

1. Abra o projeto criando no exercício anterior
2. Crie um novo projeto do tipo ClassLibrary, adicionando este projeto a solução atual
3. Mova os arquivos com as classes relativas a contas correntes para este novo assembly
4. Crie um novo projeto do tipo ClassLibrary chamada Logger. Adicione este projeto a mesma solução que estamos trabalhando
5. Mova a classe de gravação de log para este projeto
6. Crie um novo projeto do tipo ClassLibrary chamado ListaContas. Adicione este projeto a solução que estamos trabalhando
7. Mova a classe que lê os arquivos de contas para este novo componente
8. Neste ponto, o projeto original deve conter apenas a função Main. Altere o projeto para que ele faça referencia aos componentes criados nos passos anteriores. Certifique-se de que o projeto está funcionando corretamente após a alteração e que as DLLs foram geradas na pasta de compilação do projeto.

## ***Private Assemblys e Shared Assemblys***

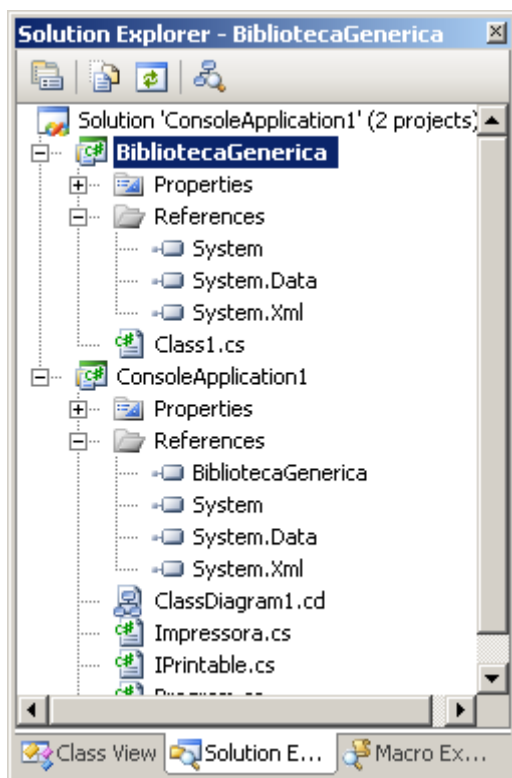
Durante o desenvolvimento de um projeto, note que quando você adiciona a referência de um assembly ao projeto e compila o projeto, o Visual Studio automaticamente copia a DLL resultante da compilação para a pasta de compilação do projeto (em geral a pasta BIN).

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.

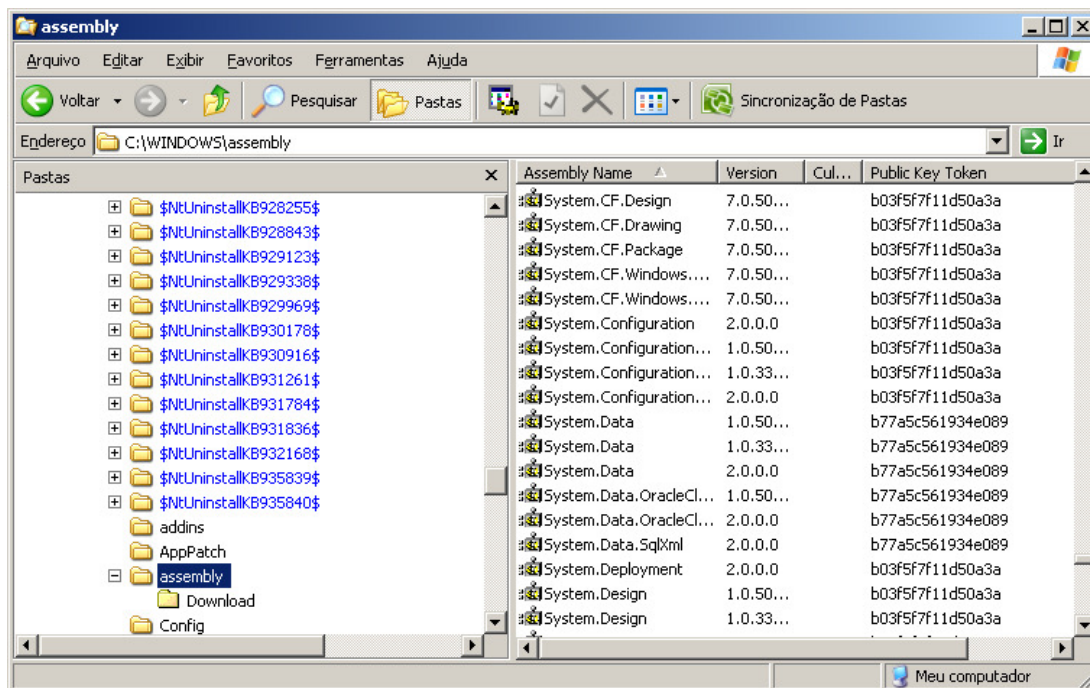
## ***Analizando as referencias***

Em um projet, 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 pois estes 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.



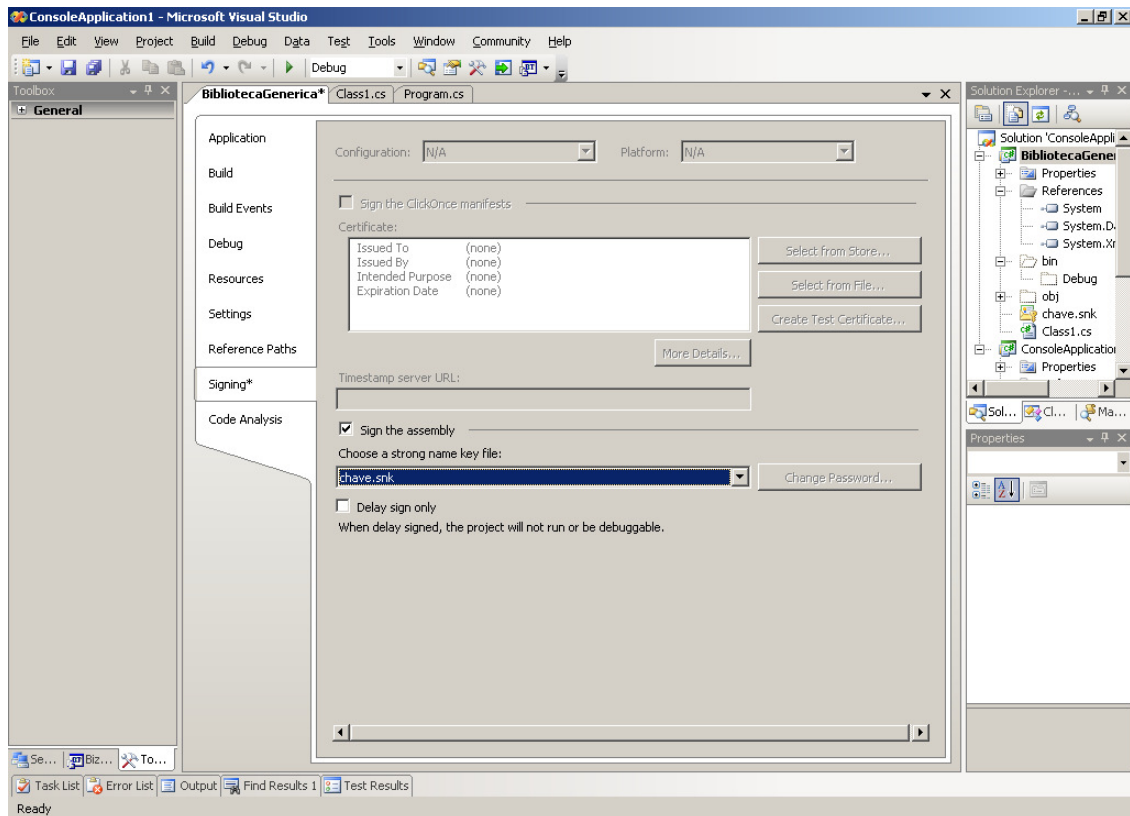
## Entendendo a estrutura do GAC

O GAC pode possuir mais de uma versão do mesmo componente. Isto porque um programa que utiliza um determinado componente está associado diretamente a versão do componente a qual ele foi compilado. Desta forma, se tivermos um componente com sua versão 1.0 e 2.0 no GAC, se um programa que faz uso destes componentes for executado, e este programa tiver sido compilado com a versão 1.0, este programa fará uso da versão 1.0 da DLL.

## Criando componentes compartilhados

Para criarmos componentes que possam ser inseridos no GAC precisamos criar uma chave para a assinatura destes componentes.

O processo de criação de chaves de assinatura é extremamente simples quando estamos trabalhando com o Visual Studio .NET. Para criarmos uma chave de assinatura, basta acessarmos as propriedades do projeto e acasarmos a opção Signing, conforme a tela a seguir.



Basta marcamos a opção “Sign the assembly” e selecionarmos a chave desejada. Podemos também criar uma chave nova através da opção “New Key File...” do combobox.

Após especificarmos a chave, basta compilar a DLL novamente e instalar a DLL no GAC.

## ***Instalando um assembly no GAC***

Para instalarmos um assembly no GAC podemos utilizar duas opções

1. O aplicativo GACUTIL
2. O Windows Explorer

Para instalarmos um componente no GAC através do GACUTIL, basta acessarmos a pasta onde se encontra esta ferramenta (que acompanha o SDK do .NET Framework) e chamarmos utilizando a seguinte sintaxe.

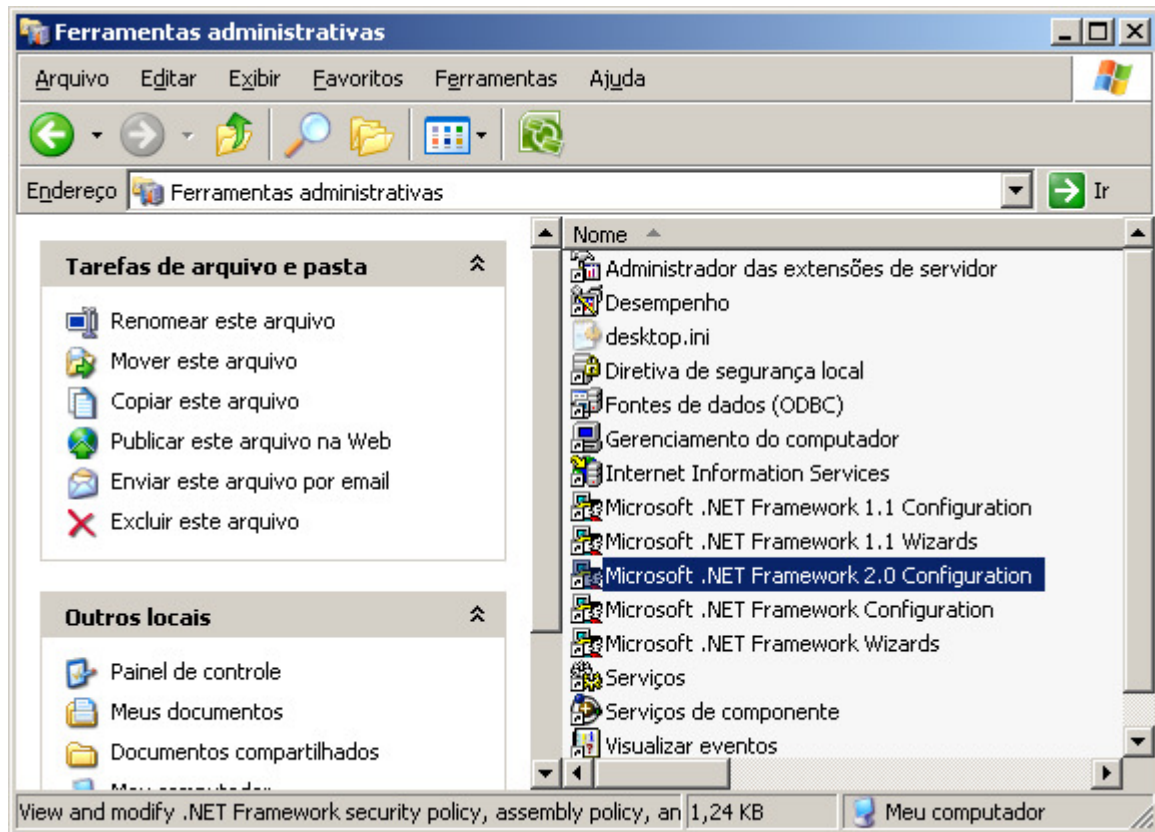
```
Gacutil -i meuassembly.DLL
```

Se quisermos instalar um assembly no GAC através do Windows Explorer, basta arrastarmos a DLL para a pasta Windows Assembly.

## ***Binding policies***

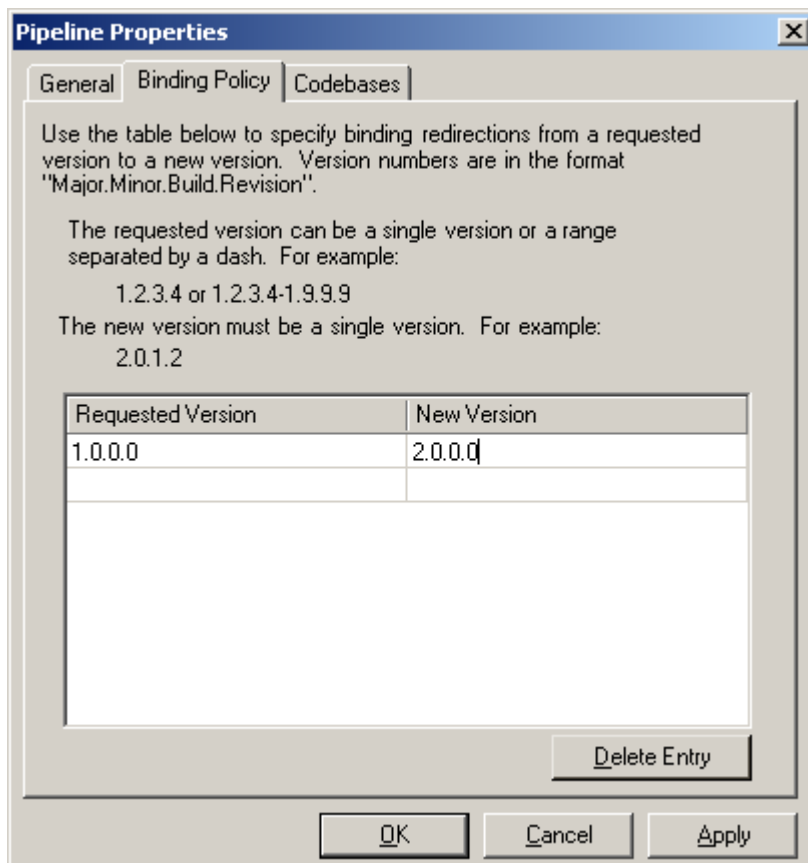
Conforme já falamos anteriormente, podemos ter mais de uma versão de um assembly instalado na máquina. Nestas situações, o programa que faz uso deste assembly utilizará a versão com o qual ele foi compilado.

Apesar deste vínculo de versão existente nos executáveis do .NET, podemos fazer com que a solicitação dos componentes seja redirecionada para outras versões. Podemos por exemplo fazer com que sempre que um programa solicitar a versão 1.0 de um assembly, ele seja direcionado a utilizar a versão 2.0, mesmo que tenha sido compilado com a versão 1.0. Podemos criar este tipo de configuração através da ferramenta de configuração do .NET Framework, que fica no painel de controle da máquina.



Através dessa ferramenta, podemos configurar as políticas de versionamento de um assembly. Para isto, basta selecionar a opção Manage Configured Assemblies, Configure an Assembly. Após a configuração do assembly, podemos fazer com que as solicitações para versões específicas do assembly sejam redirecionadas para outras versões.





## ***Exercício Prático 2 – Colocando assemblies no GAC***

1. Abra os exercícios utilizados anteriormente
2. Acesse as propriedades do componente de log e certifique-se de que ele possui a versão 1.0
3. Associe uma chave de assinatura ao componente de log chamada log.snk
4. Compile o projeto novamente
5. Adicione a DLL gerada pelo componente de log no global assembly cachê
6. Abra a pasta que contem a aplicação compilada e delete a DLL de log
7. Execute a aplicação e certifique-se de que ela está gravando log normalmente
8. Agora, abra apenas o projeto de log, e não a solução inteira
9. Altere o componente de log para que as logs sejam gravadas em arquivos contendo o seguinte formato.

LOGddMM.log

Onde dd representa o dia atual e MM o mês atual.

10. Altere a versão do componente para 2.0.0.0
11. Compile o projeto e adicione a DLL ao Global Assembly Cache novamente
12. Entre na pasta Windows Assembly e verifique que existem duas versões do componente disponíveis.

13. Vá até a pasta que contém o executável da aplicação
14. Execute a aplicação novamente e verifique que as logs não estão sendo gravadas no formato alterado
15. Altere as configurações do assembly de log, no painel de controle, para que sempre que for solicitada a versão 1.0.0.0 do assembly seja utilizada a versão 2.0.0.0.
16. Execute a aplicação novamente e verifique o resultado.

## **12. Reflection, Late Binding e Atributos**

## Introdução

Neste capítulo iremos analisar alguns conceitos avançados como Reflection, Late Binding e Atributos de compilação. Estes recursos permitem buscar informações sobre componentes e assemblies utilizados em nossos projetos, bem como a carga dinâmica de assemblies em tempo de execução.

## Entendendo Reflection

Na arquitetura .NET, reflection significa descobrir informações sobre um determinado tipo em tempo de execução. Utilizando os serviços de reflection é possível obter metadados sobre um determinado tipo, utilizando um object-model extremamente amigável. Como qualquer outro namespace, o namespace System.Reflection possui uma série de tipos, que são utilizados para esta busca de informação em tipos de dados. Alguns destes tipos podem ser observados na tabela a seguir.

Tipo	Utilização
Assembly	Esta classe possui uma série de métodos que permitem o carregamento e a manipulação de assemblies dinamicamente
AssemblyName	Esta classe nos permite obter uma série de informações sobre a identidade do assembly
EventInfo	Esta classe contém uma série de informações sobre eventos relacionados a uma classe
FieldInfo	Esta classe contém uma série de informações sobre campos de uma classe ou estrutura
MemberInfo	Esta classe contém informações sobre membros em geral. Ela serve como classe abstrata para a classe EventInfo, FieldInfo, MethodInfo e PropertyInfo
MethodInfo	Esta classe contém uma série de informações sobre métodos de uma classe ou estrutura
ParameterInfo	Contém informações sobre parâmetros de um método
PropertyInfo	Esta classe contém uma série de informações sobre propriedades de uma classe ou estrutura

## A classe System.Type

A classe System.Type possui uma série de membros que permite a análise de um determinado tipo em tempo de execução. Por exemplo, podemos utilizar o método Type.GetMethods() para obter uma lista dos métodos suportados pela classe, ou o método GetFields() para obter uma lista de todos os campos disponíveis na classe. Além disso, podemos utilizar propriedades como IsClass,

IsAbstract, IsEnum, etc. para saber mais informações sobre o tipo de objeto que o tipo representa.

Para obtermos uma referência a uma instância da classe `Ssystem.Type`, podemos utilizar o método `GetType` que está disponível na classe `System.Object`. Pelo fato deste método estar disponível na classe `System.Object`, ele se faz presente em qualquer tipo do .NET, seja ele um tipo primitivo ou um tipo customizado.

```
ContaCorrente cta = new ContaCorrente();

Type tipo = cta.GetType();
```

Outra opção é utilizarmos a função estática `GetType` da classe `Type`. Neste caso, devemos informar o nome do tipo e a biblioteca sob forma de um string.

```
Type tipo = Type.GetType("BERGS.ContaCorrente, ConsoleApplication1");
```

Ainda temos uma terceira maneira de obter uma instância da classe `Type` para um determinado tipo, utilizando a função `typeof()`.

```
ContaCorrente cta = new ContaCorrente();

Type tipo = typeof(cta);
```

## ***Obtendo metadados através da classe Type***

Obtendo uma instância da classe `type` é possível obtermos metadados sobre um determinado tipo. O exemplo a seguir utiliza o método `GetMethods()` para obter os métodos disponíveis em um determinado tipo.

```
Console.WriteLine("***** Methods *****");
MethodInfo[] mi = t.GetMethods();
foreach(MethodInfo m in mi)
    Console.WriteLine("->{0}", m.Name);
Console.WriteLine("");
```

Podemos ainda buscar informações sobre os campos disponíveis no objeto, através do seguinte trecho de código.

```
Console.WriteLine("***** Fields *****");
FieldInfo[] fi = t.GetFields();
foreach(FieldInfo field in fi)
    Console.WriteLine("->{0}", field.Name);
Console.WriteLine("");
```

Além deste métodos, ainda podemos utilizar algumas propriedades para mostrarmos certas

características do tipo.

```
Console.WriteLine("***** Fields *****");
FieldInfo[] fi = t.GetFields();
foreach(FieldInfo field in fi)
    Console.WriteLine("->{0}", field.Name);
Console.WriteLine("");
```

## ***Exercício Prático 1 – Obtendo informações sobre tipos***

1. Abra o exercício criado anteriormente
2. Na classe de log, adicione um novo objeto chamado DumpObject que receba um parâmetro do tipo Object
3. Utilizando a função GetType, obtenha as informações sobre o tipo de dados passado para a função DumpObject
4. Faça com que a função grave no arquivo de log as seguintes informações
  - a. Nome da Classe
  - b. Métodos suportados
  - c. Propriedades Suportadas
5. Na entrada do programa (função Main) faça com que um Dump do objeto que carrega as contas do arquivo seja armazenado na log.

## ***Carregando assemblies dinamicamente***

O namespace System.Reflection possui uma classe chamada Assembly. Esta classe permite que façamos a carga de um determinado componente em tempo de execução. Note que esta carga é possível mesmo não existindo uma referência direta a este componente no assembly.

No exemplo a seguir, fazemos a carga de um assembly de forma dinâmica.

```
Assembly asm = null;

asm = Assembly.Load("MyLib");

Console.WriteLine("\n***** Tipos no Assembly *****");
Console.WriteLine("->{0}", asm.FullName);
Type[] types = asm.GetTypes();
foreach (Type t in types)
    Console.WriteLine("Type: {0}", t);
Console.WriteLine("");
```

Podemos ainda carregar um assembly diretamente através de um arquivo, utilizando a função

LoadFrom.

```
Assembly asm = null;

asm = Assembly.LoadFrom("C:\\MyLIB.DLL");

Console.WriteLine("\n***** Tipos no Assembly *****");
Console.WriteLine("->{0}", asm.FullName);
Type[] types = asm.GetTypes();
foreach (Type t in types)
    Console.WriteLine("Type: {0}", t);
Console.WriteLine("");
```

## ***Entendendo Late Binding***

A técnica de late binding é uma técnica que permite a criação de um tipo em tempo de execução e a execução de seus métodos sem necessariamente ter um conhecimento prévio sobre este componente (sem ter uma referência a este componente).

A forma mais correta é implementarmos a chamada aos métodos através de early binding, ou seja, através da adição de referências ao projeto e chamadas diretas aos métodos, porém em alguns casos o late binding pode auxiliar, como por exemplo em casos em que queremos implementar algum tipo de rotina de autorização para criação de objetos.

## ***A classe System.Activator***

Para criarmos uma instância de um tipo não conhecido pelo programa, devemos utilizar a classe System.Activator. Esta classe cria uma instância de um tipo passado por parâmetro, desde que o assembly onde este tipo está contido esteja carregado.

A seguir podemos ver a utilização da classe activator para a criação de um tipo de forma dinâmica. Note que o assembly foi carregado antes da criação do objeto.

```
Assembly asm = null;

asm = Assembly.LoadFrom("C:\\MyLIB.DLL");

Type tipo = Type.GetType("MeuNamespace.MinhaClasse, MyLib");

object obj = System.Activator.CreateInstance(tipo);
```

## ***Chamando métodos sem parâmetros***

De nada adianta criarmos uma instância de uma classe dinamicamente se não podemos invocar seus métodos. Para invocarmos os métodos de uma classe de forma dinâmica podemos utilizar o método Invoke, da classe MethodInfo. Para obtermos uma instância da classe MethodInfo, basta chamarmos a função GetMethod da classe Type, conforme o exemplo a seguir.

```
Assembly asm = null;

asm = Assembly.LoadFrom("C:\\MyLIB.DLL");

Type tipo = Type.GetType("MeuNamespace.MinhaClasse, MyLib");

object obj = System.Activator.CreateInstance(tipo);

MethodInfo mi = tipo.GetMethod("MinhaFuncao");

mi.Invoke(obj, null);
```

O segundo parâmetro da classe Invoke representa os parâmetros a serem passados, que neste caso não existem.

### ***Chamando métodos com parâmetros***

Para chamarmos um método dinamicamente com parâmetros, devemos passar no segundo argumento do método invoke um array de objetos contendo os valores dos respectivos parâmetros a serem passados. O exemplo a seguir ilustra a chamada de um método que utiliza parâmetros.

```
Assembly asm = null;

asm = Assembly.LoadFrom("C:\\MyLIB.DLL");

Type tipo = Type.GetType("MeuNamespace.MinhaClasse, MyLib");

object obj = System.Activator.CreateInstance(tipo);

MethodInfo mi = tipo.GetMethod("Soma");

object[] paramArray = new object[] { 1, 2 };

object ret = mi.Invoke(obj, paramArray);
```

### ***Exercício Prático 2 – Chamando métodos dinamicamente***

1. Crie uma nova aplicação chamada testador de componentes
2. Faça com que a aplicação solicite ao usuário uma DLL para ser testada
3. Carregue a DLL em memória utilizando o método LoadFrom
4. Mostre ao usuário a lista de métodos disponíveis e permita que o usuário escolha um método para teste
5. Ao escolher um método, mostre os parâmetros necessários ao usuário



6. Solicite ao usuário que informe os parâmetros
7. Após informar o último parâmetro, execute o componente solicitado e mostre o resultado para o usuário

## ***Trabalhando com atributos***

Os atributos são elementos que geram um conjunto de metadados para o assembly. Os atributos podem ser aplicados em classe, métodos, propriedades e estruturas. Para definirmos um atributo para um destes elementos, basta informar o nome do atributo entre colchetes antes da declaração do tipo, conforme o exemplo a seguir.

```
[Serializable]
public class ContaCorrente
{
    [Obsolete]
    public void Sacar(double valor)
    {
    }

    [Conditional("DEBUG")]
    public void GravaLog()
    {
    }

    [MTAThread]
    public void Main()
    {
    }
}
```

No exemplo acima, utilizamos um atributo na classe para marcá-la como uma classe serializável (veremos mais sobre este atributo no próximo capítulo). Utilizamos também o atributo [Conditional] no método GravaLog, que já vimos anteriormente, e o atributo Obsolete no método Sacar, quer permite marcar um método como obsoleto. Também utilizamos o atributo MTAThread na função main, quer marca nosso programa para suportar múltiplas threads.

## **13. Serialização**

## Entendendo a serialização de objetos

Neste capítulo veremos o processo de serialização. O termo serialização descreve o processo de persistir (e provavelmente transferir) o estado de um objeto em um stream (conjunto de bytes). Os dados persistidos contêm todas as informações necessárias para reconstruir (ou deserializar) o estado do objeto posteriormente.

### Configurando objetos para serialização

Para tornar um objeto serializável no .net, devemos colocar o atributo [Serializable] na declaração do objeto. Se você definir que algum membro da classe não deve ser serializado, você deve marcar este membro com o atributo [NonSerialized]. Em geral utilizamos tal atributo para membros que contêm objetos que não representam o estado do objeto.

```
[Serializable]
public class Cliente
{
    public string nomeCliente;
    public string cpf;
    DateTime dataNascimento;

    [NonSerialized]
    public int idade;
}
```

No exemplo acima estamos marcando a classe cliente como serializável. O atributo idade não será serializado, pois pode ser recalculado a partir da data de nascimento.

É importante salientar que o comportamento padrão da serialização faz com que os campos públicos da classe sejam serializados. É claro que o que será serializado (campos públicos ou campos privados) depende muito do tipo de formatador utilizado. Se utilizarmos o BinaryFormatter por exemplo, todos os campos da classe serão serializados, independente de públicos ou privados. Já o XmlSerializer serializa apenas os campos públicos da classe.

### Utilizando o BinaryFormatter

Para avaliarmos como é simples o processo de serialização no .NET, vamos utilizar a classe BinaryFormatter para serializar um objeto do tipo Cliente.

A classe BinaryFormatter é composta por dois métodos principais, o Serialize e Deserialize. Para serializarmos uma instância de classe, devemos informar para o método de serialização um Stream, que pode ser um arquivo ou mesmo uma sequência de bytes em memória.

```
BinaryFormatter bin = new BinaryFormatter();

Cliente cli = new Cliente();
```

```
cli.cpf = "7777777777";  
cli.nomeCliente = "Teste";  
cli.dataNascimento = new DateTime(1978, 11, 7);  
  
Stream file = new FileStream("cliente.dat", FileMode.CreateNew);  
  
bin.Serialize(file, cli);  
file.Close();
```

O método `Serialize` aceita como parâmetro um stream onde serão armazenados os dados e uma instancia do objeto a ser serializado. O processo é extremamente simples.

Para deserializarmos um objeto, o processo é exatamente o inverso, conforme mostrado no exemplo a seguir.

```
file = new FileStream("cliente.dat", FileAccess.Read);  
  
Cliente cli = (Cliente) bin.Deserialize(file);
```

## ***Exercício Prático 1 – Serializando as classes de contas***

1. Abra o exercício utilizado nos módulos anteriores
2. Altere a definição das classes de conta para que estas classes sejam serializáveis
3. Altere a definição da classe de gravação do arquivo de contas para que esta seja serializável. As variáveis relativas ao tratamento do arquivo das contas deve ser marcada como não serializavel.
4. Altere o processo que grava o arquivo de contas para que ele salve as contas em um arquivo binário chamado `contas.DAT`. Você deve alterar a função `Main` para que ela grave as contas através do processo de serialização, ao invés de chamar os métodos de gravação de arquivos.
5. Execute a aplicação e certifique-se que o arquivo está sendo gerado
6. Altere o processo que lê as contas para que ele deserialize a classe que contém a lista de contas a partir do arquivo `contas.DAT`
7. Execute o processo novamente e certifique-se de que ele está funcionando sem problemas

## ***Utilizando o SoapFormatter***

O `SoapFormatter` funciona de forma muito semelhante ao `BinaryFormatter`, porém serializa o objeto em um formato SOAP. Este tipo de formatador é um forte candidato para quando queremos serializar os dados para o envio através da internet. A seguir temos um exemplo de serialização do

objeto cliente utilizado o SoapFormatter.

```
SoapFormatter bin = new SoapFormatter();

Cliente cli = new Cliente();
cli.cpf = "7777777777";
cli.nomeCliente = "Teste";
cli.dataNascimento = new DateTime(1978, 11, 7);

Stream file = new FileStream("cliente.xml", FileMode.CreateNew);

bin.Serialize(file, cli);
file.Close();
```

Para a deserealização, o processo também é semelhante ao BinaryFormatter, conforme o exemplo a seguir.

```
file = new FileStream("cliente.xml", FileAccess.Read);

Cliente cli = (Cliente) bin.Deserialize(file);
```

## ***Utilizando o XmlSerializer***

O XmlSerializer funciona de uma forma um pouco diferente dos outros formatadores e é utilizado para os casos em que queremos persistir um determinado objeto em formato XML. Para utilizarmos o XmlSerializer, devemos criar uma instancia da classe XmlSerializer passando como parâmetro os tipos que serão suportados para serialização.

No exemplo a seguir, temos o mesmo exemplo de serialização da classe cliente, porém utilizando a classe XmlSerializer para serialização.

```
XmlSerializer xml = new XmlSerializer(typeof(Cliente));

Cliente cli = new Cliente();
cli.cpf = "7777777777";
cli.nomeCliente = "Teste";
cli.dataNascimento = new DateTime(1978, 11, 7);

Stream file = new FileStream("cliente.xml", FileMode.CreateNew);

xml.Serialize(file, cli);
file.Close();
```

A deserealização funciona da mesma forma, utilizando a mesma classe e o método Deserialize.

```
file = new FileStream("cliente.xml", FileAccess.Read);
```

```
Cliente xml = (Cliente)bin.Deserialize(file);
```

É possível controlar o processo de serialização da classe XmlSerializer indicando atributos em alguns membros da classe. Podemos por exemplo indicar que o atributo deverá ser serializado como um atributo, utilizando a classe [XmlAttribute] ou como um elemento, utilizando a classe [XmlElement].

O exemplo a seguir mostra uma alteração na classe Cliente para que os membros sejam armazenados como atributos e não como elementos.

```
[Serializable]
public class Cliente
{
    [XmlAttribute]
    public string nomeCliente;~
    [XmlAttribute]
    public string cpf;
    [XmlAttribute]
    public DateTime dataNascimento;

    [NonSerialized]
    public int idade;
}
```

### ***Exercício Prático 1 – Utilizando serialização***

1. Abra o projeto utilizado para testar tipos dinamicamente, utiliza no exercício do capítulo anterior
2. Altere o projeto para que o tipo retornado dinamicamente pelo método seja serializado em Xml para uma classe do tipo StringWriter. Esta classe recebe um StringBuilder como parâmetro e grava o conteúdo em uma string utilizando um Stream
3. Apresente o Xml gerado para o usuário na saída do método
4. Execute a aplicação e verifique o resultado.

## **14. Acessando dados com ADO.NET**

## ***O que é o ADO.NET ?***

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

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

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

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

## ***Os namespaces relacionados ao ADO.NET***

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

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

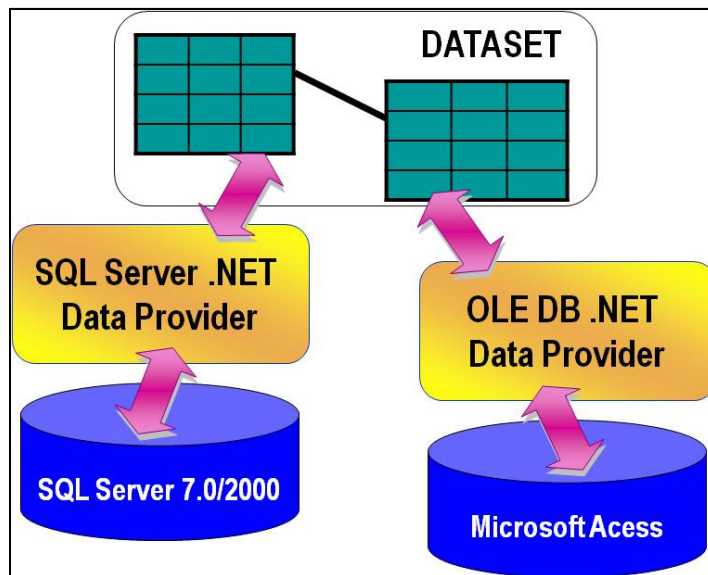
## ***O modelo de execução do ADO.NET***

O ADO.NET provê uma estrutura de acesso a dados que permite o acesso a múltiplas bases



de dados simultaneamente. É possível armazenar duas tabelas de diferentes bancos de dados (SQL Server e Access por exemplo) em uma mesma estrutura de dados (DataSet).

A estrutura responsável pelo armazenamento dos dados é o dataset, que contém um conjunto de objetos (DataTables) que representam resultados tabulares extraídos da base de dados. Estes resultados podem ser extraídos de tabelas de bancos de dados diferentes.



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

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

### ***O modelo de execução em um ambiente conectado***

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

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

**XxxConnection:** É o objeto utilizado para estabelecer a conexão com o banco. É necessário informar os parâmetros de conexão e abrir a conexão com o banco. Exemplos dessa classe são SqlConnection e OleDbConnection.

**XxxCommand:** É um objeto utilizado para enviar comandos a base de dados. É necessário montar a cláusula Sql desejada e informar ao objeto de comando. Ao executar o comando, este pode devolver um objeto do tipo XxxDataReader.

**XxxDataReader:** É um objeto utilizado para ler dados de um comando executado. O método `Read` lê os dados de registro em registro. Após a leitura completa dos dados é necessário fechar o `DataReader` e a conexão.

## ***O modelo de execução em um ambiente desconectado***

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

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

1. É aberta uma conexão utilizando um objeto `XxxConnection` (`OleDbConnection` ou `SqlConnection` por exemplo)
2. É criado um objeto do tipo `XxxDataAdapter` que é responsável por fazer a extração de dados do banco de dados para a memória e o posterior envio dos dados da memória para o banco de dados.
3. Utilizando o método `Fill`, extraímos os dados da base de dados e armazenamos em um `DataSet`. Neste momento fechamos a conexão com o banco pois os dados já estão na memória da aplicação para serem manipulados.
4. Como os dados estão em memória, é possível inserir, remover ou alterar registros do `DataSet`.
5. Ao finalizar as alterações, restabelecemos a conexão com o banco de dados para enviar as alterações
6. Utilizando o método `Update` do `DataAdapter`, enviamos as alterações para o banco de dados. O `DataAdapter` verifica os tipos de alterações que foram realizadas e executa o comando correspondente no banco de dados (inserção, exclusão, atualização).
7. Ao finalizar o processo, fechamos a conexão com o banco de dados.

## ***Estabelecendo uma conexão com um banco de dados***

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

No caso do `Sql Server`, a classe utilizada para estabelecer a conexão com o banco de dados é a `SqlConnection`. O nome destas classes varia de acordo com o `.NET Data Provider` utilizado, no caso do `OleDb` por exemplo utilizamos a classe `SqlConnection`.

Ao criarmos uma instancia da classe `SqlConnection`, devemos informar uma string de conexão, que contém todos os parâmetros para a conexão com o banco de dados, como usuário e senha.

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

```
string sConnString;  
sConnString = "Data Source=MYSERVER;IntegratedSecurity=SSPI;";  
sConnString += "InitialCatalog=NorthWind"  
SqlConnection oConn = New SqlConnection(sConnString);
```

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

Nome do Parâmetro=Valor do Parâmetro

Os parâmetros mais comuns para as strings de conexão podem ser observados na tabela a seguir.

<b>Data Source</b> - ou - <b>Server</b>	Indica o nome do servidor
<b>Integrated Security</b> - ou - <b>Trusted Connection</b>	Indica se será utilizada autenticação integrada ou não. É utilizada pelo Sql Server
<b>Initial Catalog</b> - ou - <b>Database</b>	Indica o banco de dados inicial
<b>User Id</b>	Indica o usuário para conexão
<b>Password</b>	Indica a senha do usuário
<b>Provider</b>	Indica o provedor de dados a ser utilizado. Este parâmetro só deve ser informado quando o objeto utilizado por OleDbConnection

## ***Criando comandos***

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

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

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

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

```
SqlCommand oCmd = New SqlCommand("UPDATE Products SET UnitPrice=UnitPrice*1.1");

oCmd.Connection = oConn;

oCmd.CommandText = "UPDATE Products SET UnitPrice=UnitPrice*1.05";
```

## ***Executando comando***

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

- ExecuteNonQuery: Para comandos que não executam queries
- ExecuteScalar: Para comandos que executam resultados escalares
- ExecuteReader: Para comandos que retornam conjuntos de dados

## **O método ExecuteNonQuery**

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

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

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

```
SqlCommand oCmd = new SqlCommand("UPDATE Products SET
UnitPrice=UnitPrice*1.1");

oCmd.Connection = oConn;
oCmd.CommandText = "UPDATE Products SET UnitPrice=UnitPrice*1.05";

oCmd.ExecuteNonQuery();
```

## **O método ExecuteScalar**

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

Este comando pode retornar qualquer tipo de dado.

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

```
SqlCommand oCmd = new SqlCommand("UPDATE Products SET
UnitPrice=UnitPrice*1.1");

oCmd.Connection = oConn;
oCmd.CommandText = "SELECT COUNT(*) FROM Products";

int iNumProdutos;
iNumProdutos = oCmd.ExecuteScalar();
```

## O método ExecuteReader

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

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

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

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

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

```
SqlCommand oCmd = new SqlCommand("SELECT ProductName, ProductId FROM Products",
oConn);
SqlDataReader oDr;

oDr = oCmd.ExecuteReader();

while (oDr.Read()) {
    Debug.WriteLine(oDr("ProductName").ToString());
}
```

## Passando parâmetros

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

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

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

```
SqlConnection oConn = new SqlConnection(sConnString);
SqlCommand oCmd = new SqlCommand();

oCmd.Connection = oConn;

oCmd.CommandText = "UPDATE Products " + " SET UnitPrice=@UnitPrice " + " WHERE ProductId=@ProductId";

SqlParameter oParam = new SqlParameter("@UnitPrice", 1.5);
oCmd.Parameters.Add(oParam);

oParam = new SqlParameter();
oParam.ParameterName = "@ProductId";
oParam.DbType = DbType.Int32;
oParam.Value = 1;
oCmd.Parameters.Add(oParam);

oCmd.ExecuteNonQuery();
```

### ***Exercício Prático 1 – Executando comandos na base de dados***

1. Crie um novo projeto com o nome de ImportarContas
2. Faça com que o projeto leia o arquivo Contas.TXT e insira as informações na base de dados local de sua máquina, no SQL Express. Os dados deverão ser armazenados na tabela Contas
3. Execute o programa e verifique que as contas foram importadas com sucesso. Você pode acessar a base de dados através do Server Explorer do Visual Studio.NET

### ***O que é um DataSet ?***

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

Para extrairmos dados da base de dados e preenchermos o DataSet utilizamos a classe

DataAdapter. Esta classe é capaz de executar os quatro comandos básicos de um banco de dados (Insert, Update, Delete, Select) sendo capaz de executar estas operações sobre os dados do dataset.

## ***O que é um DataAdapter ?***

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

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

Após extrairmos os dados para o DataSet, podemos modificar estes dados (que estão armazenados em memória). À medida que modificamos os dados do DataSet, este faz uma marcação nas alterações que fazemos, marcando as linhas inseridas como inseridas, modificadas como modificados e excluídos como excluídos. Quando concluímos as alterações, é possível chamar o DataAdapter novamente para que ele execute para cada linha modificada o comando correspondente a modificação realizada.

## ***Criando um DataSet e um DataAdapter***

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

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

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

## ***Criando e preenchendo um DataSet***

Para criar um novo DataSet basta utilizar o comando New e criar um novo objeto.

Para preencher um dataset utilizando um DataAdapter, devemos utilizar o método Fill do DataAdapter, informando o DataSet e nome da tabela a ser criada no DataSet.

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

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

## ***Exercício Prático 2 – Utilizando um DataSet***

1. Abra o exercício criado anteriormente que tem a tela para gerenciamento de contas correntes
2. Remova o processo de Deserialização e Serilização da classe de lista de contas
3. Altere o projeto para que as contas passem a ser lidas e gravadas na base de dados do SQL Express
4. Altere a função Main para chamar os métodos necessários para a gravação e leitura dos dados no SQL Express
5. Não esqueça de implementar tratamento de erros nas classes que manipulam a base de dados. Uma boa prática é utilizar o comando try...catch... finally e colocar o fechamento da conexão com a base de dados no bloco finally.



## **15. Introdução ao ASP.NET**

## ***O que é o ASP.NET ?***

O ASP.NET é uma evolução do antigo ASP. Assim como seu antecessor, o ASP.NET permite a construção de páginas Web dinâmicas que acessam recursos do servidor.

Diferente do ASP, o ASP.NET possui uma estrutura bem mais flexível e organizado, por utilizar todo o conceito de orientação a objetos no desenvolvimento das aplicações.

Além de ser mais flexível, o ASP.NET também é bem mais robusto e rápido, por não ser uma linguagem interpretada e sim compilada. Toda a página criada em .NET possui um módulo compilado correspondente.

O ASP.NET trabalha com o conceito de formulários Web, que é muito semelhante ao conceito de construção de aplicações com formulário para Windows, utilizado pelo Visual Basic 6.0. Através do ASP.NET podemos criar formulário que geram código HTML para o browser.

O .NET Framework verifica a linguagem em que a página foi escrita e cria uma versão compilada da mesma.

O código compilado é armazenado em uma DLL, que fica guardada em um cache temporário. Esta DLL é carregada em memória e a página é então executada. O resultado HTML gerado pela execução da página é devolvido para o browser do cliente.

Nas chamadas subseqüentes a esta página, o framework utilizara a versão da página compilada, o que evita uma re-interpretação de todo o código.

## ***O que é um WebForm ?***

Os formulários web são formulários dinâmicos processados no servidor que geram resultados HTML para o cliente. Os formulários Web possuem um conceito semelhante ao conceito dos formulários desenvolvidos para windows, com a diferença que estes formulários tratam seus eventos no servidor.

Todo o formulário Web possui a extensão aspx, e fica armazenado no servidor web.

Além da extensão, todo o formulário web possui uma diretiva de página representada pela tag a seguir:

```
<%@ Page Language="cs" %>
```

Além da diretiva de página, os webforms possuem outras tags também utilizadas em páginas asp, porém com alguns atributos adicionais.

A tag form também possui alguns atributos adicionais, como o atributo runat="server" que indica que este formulário será processado no servidor, permitindo que o ASP.NET realize o processamento e identifique os controles contidos nesse formulário.

É importante salientar que um Web form possui uma tag form com o atributo runat="server", pois a arquitetura do ASP.NET considera cada página como um formulário, não sendo possível criar mais de um formulário com referencia no servidor para a mesma página. Apesar disso, é possível criar formulários convencionais nestas páginas (sem o atributo runat="server").

### ***Bibliotecas de classe utilizadas***

Para criarmos aplicações ASP.NET, o .NET Framework disponibiliza uma série de bibliotecas que contém toda a infra-estrutura necessária para a criação destas aplicações.

Dentre as principais bibliotecas, encontramos as bibliotecas a seguir:

- System.Web: Contém a infra-estrutura básica para a construção de aplicações ASP.NET
- System.Web.Services: Contém a intra-estrutura básica para a construção de web services
- System.Web.UI: Contém a infra-estrutura para o desenvolvimento de aplicações com interface visual (HTML).

### ***Entendendo os WebForms***

Os Web Forms do ASP.NET funcionam de uma forma muito diferente das antigas páginas ASP. Dentre as características que encontramos nos WebForms, podemos encontrar as seguintes:

**Processamento:** Os webforms não possuem um processamento seqüencial como as páginas ASP.

**Atributos:** Além dos atributos normais, encontramos alguns atributos especiais nas tags dos webforms. Um destes atributos é o atributo runat="server". Este atributo indica que o formulário possui uma referência no servidor, e que deve ser processado de acordo com as regras do ASP.NET. Além do atributo runat="server", o formulário também possui uma identificação ("id") que identifica o formulário no servidor.

**Eventos:** Todo o processamento é feito por meio de objetos e eventos. A página é montada em resposta aos eventos que ocorrem durante o processamento.

**Controles Web:** Os elementos do formulário que possuem uma referência no servidor são chamados de "controles web". Este controles permitem a interação e a montagem dinâmica da página.

**Orientação à Objetos:** Assim como os controles Web, o próprio formulário Web é

considerado uma classe. É a partir da implementação desta classe que criamos o funcionamento do nosso Web Form.

A seguir temos um exemplo da seção HTML e do código fonte de um Web Form:

```
<%@ Page Language="cs" %>
<HTML>
  <body>
    <form id="Form1" method="post" runat="server">
      <asp:TextBox ID=txtNome Runat=server></asp:TextBox>
      <asp:Button id="Button1" runat="server"
        Text="Button"></asp:Button><BR>
      <INPUT id="Checkbox1" type="checkbox"
        name="Checkbox1"
        runat="server" EnableViewState="True">
    </form>
  </body>
</HTML>
<Script runat=server>
  Sub Page_Load
    txtNome.Text = "Aluno !"
  End Sub
</Script>
```

### ***Exercício Prático 1 – Criando um projeto Web***

1. Crie um novo projeto do tipo WebSite chamado Banco
2. Adicione uma referencia no projeto apontando para as DLLs criadas nos exercícios anteriores (Bancos, Lista de Contas e Logs)
3. Adicione um título a página Default.ASXP criada no projeto
4. Execute a aplicação e verifique o resultado

### ***O que são controles Web ?***

Quando montamos páginas HTML, criamos uma série de tags para dar forma a esta página. Os controles web asp.net são na realidade elementos da página HTML (tags) que possuem uma referencia no servidor. Esta referencia permite que possamos interagir com estes controles (tags) mudando suas propriedades e características.

Para se tornar um controle web, uma tag precisa de dois atributos, o atributo runat="server" e o atributo id, que indica o nome do controle web no servidor.

Uma das características que os controles web possuem é a manutenção de estado, ou seja, a

medida que a página é enviada para o servidor para processamento e retorna, os valores dos controles web não são perdidos (como no modelo do asp por exemplo).

As características dos controles web são todas controladas pelo servidor e podem ser alteradas no tratamento dos eventos que ocorrem durante o processamento da página.

Outra característica dos controles web é que eles geram código específico para o browser que o client está utilizando. Se um controle web envia um bloco de script para o browser por exemplo, ele pode possuir uma funcionalidade especial para browsers que não suportam código script.

## ***A estrutura de um controle Web***

Os controles web, assim como qualquer outro elemento do formulário web é representado por uma tag. A diferença é que esta tag pode possuir uma sintaxe diferenciada. Mesmo com esta sintaxe, após o processamento, este controle é transformado em HTML.

As tags que representam os controles web, possuem duas características principais, o atributo `runat="server"` que indica que o controle possui uma referência no servidor, e o atributo `Id`, que identifica unicamente o controle dentro do formulário.

Um exemplo de controles web pode ser observado a seguir

```
<asp:Button
    id="Button1"
    runat="server"
    Text="Button"></asp:Button>

<INPUT id="Checkbox1"
    type="checkbox" name="Checkbox1" runat="server">
```

## ***A referência no servidor***

A referência no servidor se dá a partir da criação de variáveis criadas na classe que representa o formulário web na página code-behind. Esta variável deve possuir o mesmo nome do `Id` indicado para o controle na página de layout (aspx).

O tipo da classe tem uma correspondência a uma classe do .NET Framework, que indica a funcionalidade básica dos controles.

Como estas variáveis possuem um escopo de classe, elas são visíveis em qualquer método implementado dentro da classe do formulário.

Um exemplo de declaração de dois controles Web pode ser observado a seguir.

```
protected System.Web.UI.WebControls.Button Button1;

protected System.Web.UI.HtmlControls.HtmlInputCheckBox Checkbox1;
```

## ***Propriedades dos controles web***

Os controles web possuem as seguintes características:

- **Manutenção de estado (ViewState):** Todo o controle web possui manutenção de estado, ou seja, seu estado interno se preserva quando a página é enviada ao servidor e retorna;
- **Referência no servidor:** Todo o controle web possui uma referência no servidor, o que permite a interação e alteração de suas características;
- **Propriedades e Métodos:** Os controles web possuem propriedades e métodos que permitem a interação e a troca das características dos controles;
- **Eventos:** Além de propriedades e métodos, os controles web possuem eventos nos permitem tomar ações de acordo com as interações do usuário com nossos controles.
- **Geração de código HTML:** Todo o controle web geram conteúdo HTML para o client.

## ***WebServer Controls***

Os controles web podem ser divididos em duas categorias, os Web Server controls e os HTML Server Controls.

Os Web Server controls são controles que possuem uma tag diferenciada. Apesar de possuírem esta tag, o resultado de seu processamento é sempre HTML, ou seja, eles sempre enviando HTML para o browser.

Os controles web podem ser divididos nas seguintes categorias:

**Intrínsecos:** São controles que possuem uma correspondência as tags HTML convencionais (Caixa de Texto, Botão, Caixa de seleção, etc.)

**Rich Content:** Possuem conteúdo e funcionalidade ricas, como um calendário por exemplo.

**Vinculados:** São controles que podem ser vinculados a fontes de dado, como tabelas de bases de dados.

**De Validação:** São controles que permitem a validação dos campos do formulário

**Do Internet Explorer:** São controles com funcionalidades especiais que são funcionam no internet explorer.

## ***Html Server Controls***

Os HTML Server controls são controles que possuem tags iguais as do HTML convencional, porém com o atributo runat=server. Qualquer tag HTML pode ser transformada em uma HTML Server control, bastando para isso adicionar o atributo runat=server e um atributo id.

Os HTML Server Controls possuem gerenciamento de estado, assim como qualquer controle

web.

Por serem controles gerados a partir de tags HTML, podemos ter um controle maior sobre o HTML final gerado para o usuário.

Assim como os Web Server Controls, os HTML server controls possuem classes no framework que implementam sua funcionalidade básica.

O object model dos HTML server controls é muito semelhante ao do Javascript, ou seja, as propriedades e métodos disponíveis nos controles é extremamente parecida as encontradas no modelo utilizado nas linguagens client (javascript e vbscript).

## ***Exercício Prático 2 – Criando uma aplicação ASP.NET***

1. Abra o exercício criado anteriormente
2. Adicione um controle do tipo Menu da barra de ferramentas
3. Acesse a propriedade Items do menu na janela de propriedades e adicione os seguintes itens de menu

Nome do Menu	Navigate URL
Visualizar Logs	viewLogs.aspx
Saldo	Saldo.aspx

4. Adicione dois web forms ao projeto, um chamado viewLogs.aspx e outro chamado buscarConta.aspx
5. Coloque um título em cada um dos formulários criados
6. Execute a aplicação e verifique se o menu está desviando o usuário para a página correta
7. Adicione um controle do tipo GridView na página listarContas.aspx
8. Acesse a smart tag do controle (flecha que fica no canto direito e no tipo do componente).
9. Na opção data source, selecione New Data Source e selecione a opção SQL
10. Clique em next e configure componente para que as contas sejam listadas a partir da base de dados do SQL Server Express

## **16. Construindo aplicações em ASP.NET**



## ***Entendeno o gerenciamento de estado***

Por gerenciamento de estado entende-se a capacidade da aplicação manter o estado das informações durante a navegação do usuário em duas páginas.

O gerenciamento de estado pode aparecer em três níveis

- Página
- Sessão
- Aplicação

### ***Gerenciando o estado da página***

O gerenciamento de estado da página é a capacidade da página manter os seus valores a que vez que ela vai até o servidor para o processamento e reporta.

O responsável pelo gerenciamento do estado da página é o ViewState, que é gerenciado automaticamente pelo asp.net. O Viewstate é representado por um campo escondido, que é enviado para a página toda vez que ela é processada.

A maior parte dos controles web que utilizamos faz uso do ViewState e seu tamanho pode variar de acordo com o número de controles que utilizamos no formulário.

### ***Interagindo com o ViewState***

É possível interagir com o viewstate criando informações adicionais em seu conteúdo. Para isto, basta criar uma nova variável através do objeto viewstate. Para criar uma nova variável, é só informar o nome da mesma entre parênteses para o objeto viewstate e atribuir o seu valor.

A leitura de variável é feita da mesma forma. Um exemplo de utilização do ViewState pode ser observado a seguir.

```
ViewState["MinhaVariavel"] = "Meu Conteudo";  
  
string sMeuConteudo;  
sMeuConteudo = ViewState["MinhaVariavel"].ToString();
```

É importante lembrar, o viewstate é válido somente no processo de postback da página. Ao sairmos da página para uma outra página da aplicação, o ViewState se perde.

### ***O que são Cookies ?***

Cookies são informações textuais enviadas do cliente para o server e do server para o cliente através do header http. Cada vez que requisitamos e recebemos uma página, um cookie pode acompanhar o header http da requisição.

Os cookies são gerenciados pelo navegador (browser) e podem ser de dois tipos:

- Temporários (session cookies)
- Com expiração (se mantém até a data de expiração)

## ***Criando cookies temporários***

Para criarmos um cookie temporário, devemos criar uma nova instancia da classe `HttpCookie`, disponível no framework. Ao criar esta nova instancia, devemos informar o nome do cookie.

Um cookie pode conter um valor único ou múltiplos valores. Para atribuir um valor único para o cookie, basta atribuir um valor para a sua propriedade `Value`. Para criar múltiplos valores, basta inserir os mesmo na coleção `Values` do `Cookie`.

Ao finalizar a criação do cookie, basta adiciona-lo a coleção `Cookies` do objeto `Response` que ele será automaticamente enviado para o cliente.

Um exemplo de criação de um cookie temporário pode ser visualizado a seguir.

```
HttpCookie oCookie = new HttpCookie("MEU_COOKIE");

oCookie.Value = "Valor Único no Cookie";
oCookie.Values["HORA_LOGON"] = Now.ToString();
oCookie.Values["NOME_VISITANTE"] = Request("Nome");

Response.Cookies.Add(oCookie);
```

## ***Lendo informações de um Cookie***

Para ler as informações de um cookie armazenado no client, podemos acessar a coleção `cookies` do objeto `Request`.

Através da coleção `cookie`, podemos um obter um cookie através de seu nome e ler os seus valores.

Um exemplo desta implementação pode ser observado a seguir.

```
HttpCookie oMeuCookie = Request.Cookies["MEU_COOKIE"];

string sMeuValor = oMeuCookie.Value;
DateTime dtHora = oCookie.Values["HORA_LOGON"];
```

## ***O que são variáveis de Sessão ?***

As variáveis de sessão são informações armazenadas pela aplicação no servidor para a sessão de um determinado usuário. Cada usuário possui suas próprias variáveis, e estas são destruídas no momento em que a sessão do usuário se encerra.

## ***Como funciona o gerenciamento de Sessão ?***

O gerenciamento da sessão funciona da seguinte forma:

- 1) O usuário requisita uma página do servidor
- 2) O ASP.NET identifica que o usuário não possui uma sessão estabelecida com o servidor
- 3) O ASP.NET gera uma nova sessão para este usuário e gera uma chave de sessão (Session\_Id)
- 4) A página é processada e juntamente com ela é enviado um cookie com o Session\_Id do usuário
- 5) Nas próximas requisições, o usuário irá enviar o cookie com o mesmo session\_id, permitindo que o ASP.NET identifique sua sessão.

## ***Interagindo com variáveis de sessão***

Para interagirmos com as variáveis de sessão, devemos utilizar o objeto Session. Para atribuir um valor a uma variável de sessão, basta informar o nome da variável entre parênteses para o objeto session que a variável será criada. A leitura é feita da mesma forma.

Um exemplo de criação de uma variável de sessão pode ser observado a seguir.

```
Session["Nome_Usuario"] = txtNomeUser.Text;  
Session["ID_Usuario"] = oDR["Cod_Usuario"];  
  
string sNomeUser = Session["Nome_Usuario"].ToString();
```

## ***O controle de sessão***

A sessão do usuário com o ASP.NET pode ser customizada através de parâmetros encontrados no web.config. Podemos alterar informações como o tempo de expiração e o modo de funcionamento da sessão através das propriedades do elemento sessionState do web.config.

O atributo timeout indica o tempo de duração da sessão em minutos. Um exemplo de customização de sessão no arquivo web.config pode ser observado a seguir.

```
<sessionState  
    mode="InProc"  
    stateConnectionString="tcpip=127.0.0.1:42424"  
    sqlConnectionString="datasource=127.0.0.1;  
                        Trusted_Connection=yes"  
    cookieless="false"  
    timeout="20"  
/>
```

## ***Modos de armazenamento de sessão***

O ASP.NET permite o armazenamento das informações de sessão do usuário de várias formas. Os diferentes modos de armazenamento de sessão permitem tornar a aplicação mais escalável.

Os modos de armazenamento de sessão disponíveis são:

InProc: Na memória do Servidor

StateServer: Servidor de gerencia de estado

SqlServer: Banco de dados

## ***Considerações sobre escalabilidade***

Para tornar a aplicação mais escalável, podemos atribuir o gerenciamento das variáveis de sessão a um servidor somente. Desta forma podemos ter um parque de servidores web para atender o processamento das páginas enquanto que as variáveis de sessão são armazenadas em outro servidor. Isto permite que a requisição do usuário possa ser atendida por qualquer um dos servidores web, pois suas variáveis de sessão estarão armazenadas não na memória do servidor, e sim em um servidor externo.

## **Iniciando o StateServer**

Uma das opções que temos para armazenamento de sessão é o state server. O state server é um servidor instalado com o .NET Framework que gerencia os dados de sessão de um servidor ASP.NET. Para utilizarmos um stateserver, basta alterarmos o modo de gerenciamento de sessão no web.config para state server e iniciar o serviço de gerenciamento de estado do servidor que será o state server.

## ***Variáveis de aplicação***

As variáveis de aplicação são variáveis disponíveis para a aplicação como um todo, independente da sessão do usuário que estiver acessando o conteúdo da variável.

A criação destas variáveis é feita exatamente da mesma forma que as variáveis de sessão, porém utilizando o objeto application.

Por estarem disponíveis para todos os usuários da aplicação, esta variável necessita controle de concorrência em sua gravação. Sempre que vamos gravar uma variável de aplicação, devemos chamar o método Lock. Ao gravar o valor, devemos chamar o método Unlock.

Um exemplo de criação de uma variável de aplicação pode ser observado a seguir.

```
Application.Lock()  
Application("NumVisitantes") = _  
    Convert.ToDouble(Application("NumVisitantes")) + 1  
Application.Unlock()
```

## ***O arquivo global.asax***

O arquivo global.asax gerencia os eventos globais da aplicação. Através dele podemos criar variáveis na inicialização da aplicação ou da sessão do usuário.

O Global.ASAX também possui eventos para mapear as requisições feitas a aplicação, além de possibilitar o tratamento de erros ocorridos em toda a aplicação.

## **Eventos do global.asax**

Os eventos mais importantes disponíveis no arquivo global.asax são:

- Session\_OnStart
- Session\_OnEnd
- Application\_OnStart
- Application\_OnEnd
- Application\_Error
- Application\_BeginRequest

## ***Exercício Prático 1 – Utilizando cookies, variáveis de sessão e aplicação***

1. Abra a aplicação criada no exercício anterior
2. Altere a aplicação para que seja criado um cookie ao acessar a página Default.ASPX. No cookie deverá ser armazenada a data e hora do último acesso do usuário
3. Crie um novo controle do tipo Label na página Default.ASPX
4. Leia o valor do cookie armazenado e exiba para o usuário a data e hora do seu último login na label criada. Faça este processo antes de gravar as informações de novas ao cookie
5. Ainda na página default, crie um controle do tipo TextBox e outro do tipo Button. Coloque um texto ao lado do text Box com os dizeres: “Informe o número de sua Conta”
6. Implemente o evento de click no botão para que a conta do usuário seja lida utilizando as classes criadas nos exercícios anteriores
7. Armazene a conta obtida em uma variável de sessão chamada CONTA

8. Desvie o usuário para a página Saldo.ASPX utilizando o seguinte comando

```
Response.Redirect("Saldo.ASPX");
```

9. Na página Saldo.ASPX, crie uma nova Label que exiba o saldo da conta selecionada. Busque a conta através da variável de sessão criada nos passos anteriores.

## ***O modelo de configuração do ASP.NET***

Ao contrário da versão anterior, o ASP.NET possui inúmeros recursos de configuração.

A configuração das aplicações ASP.NET é feita através de arquivos de configuração. Estes arquivos possuem configurações relativas ao servidor como um todo, como também configurações referentes apenas a uma aplicação web específica.

Todos os arquivos de configuração possuem um formato XML, com um número de tags definidas que representam as opções de configuração disponíveis.

O uso de arquivos de configuração em uma aplicação ASP.NET não é obrigatório, mas é altamente recomendável.

### ***O arquivo Machine.Config***

O arquivo Machine.Config contém as relativas a máquina como um todo.

Neste arquivo encontramos os valores default que se aplicam a todas as aplicações ASP.NET que se encontram na máquina em questão.

Além destas informações, encontramos no elemento Process Model algumas informações relativas ao processo que executa e processa as páginas ASP.NET. Podemos, por exemplo, alterar o contexto do usuário que executa a aplicação ASP.NET para um outro usuário (de um domínio por exemplo). Para isto basta alterar a propriedade Username e Password do elemento ProcessModel do machine.config.

### ***O arquivo Web.Config***

O arquivo Web.Config possui informações referentes a uma pasta de uma aplicação Web.

Neste arquivo encontramos configurações relativas a uma aplicação web ou a uma pasta de uma aplicação web especificamente.

Dentro deste arquivo podemos definir parâmetros de configuração como:

- Configuração do TRACE

Configurações de Cultura

- Configurações de autenticação e autorização

- Linguagem default para compilação

- Modelo de tratamento de erros

## ***Hierarquia de configurações***

Os arquivos de configuração possuem uma relação totalmente hierárquica, ou seja, alguns parâmetros de configuração criados em um nível de configuração superior podem ser sobrescritos em um nível inferior.

Um exemplo disso são as configurações encontradas no machine.config. Dentro deste arquivo encontramos os settings default para uma aplicação ASP.NET, como configuração de trace, modelo de autenticação e etc. Se estas mesmas configurações forem colocadas em um arquivo Web.Config de uma aplicação, para esta aplicação valerem os valores do Web.Config da aplicação e não do machine.config.

## ***Criando itens de configuração***

Além dos parâmetros de configuração padrão do ASP.NET, podemos criar nossos próprios parâmetros de configuração dentro dos arquivos de configuração.

Para criarmos estes parâmetros, devemos criar um elemento do tipo appSetting dentro do elemento configuration do arquivo.

Para criarmos itens de configuração, basta criarmos elementos do tipo add. Neste elemento deve ser indicado um atributo key, que representa o nome do valor que queremos armazenar, e um atributo value, que contém o valor propriamente dito.

Um exemplo de um arquivo de configuração pode ser observado a seguir.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ConnString" value="Data Source=..."></add>
  </appSettings>
  <system.web>
    ...
  </system.web>
</configuration>
```

## ***Lendo informações de configuração***

As informações do arquivo de configuração podem ser acessadas através da classe AppSettings do framework. Esta classe encontra-se dentro do namespace System.Configuration.

Para acessarmos um valor do arquivo de configuração, basta informarmos para a classe appSetting o nome da chave do item de configuração criado.

Um exemplo de leitura de uma connection string de um arquivo de configuração pode ser observado a seguir.

```
sConnString = ConfigurationSettings.AppSettings["ConnString"]
```

## ***Exercício Prático 2 – Os recursos de configuração do ASP.NET***

1. Altere o arquivo de configuração web.config da aplicação criada no exercício anterior para que a string de conexão com o banco de dados seja armazenada no web.config
2. Altere o componente que lê a lista de contas da base de dados para que obtenha a string de conexão utilizando as classes de configuração do .NET
3. Execute a aplicação e verifique o resultado



## **17. Construindo Xml Web Services**

## ***O que são web services ?***

Web Services são serviços que disponibilizam informações através da internet. Podemos imaginar um web services qualquer serviço que disponibilize algum tipo de informação como cotações da bolsa ou até mesmo previsão do tempo.

## ***Como o .NET torna os web services possíveis ?***

O ASP.NET torna os webservices possíveis através de uma infra-estrutura de classes que implementam parte da lógica necessária para a construção e o acesso aos web services.

Dentre as bibliotecas de classe utilizadas, podemos citar:

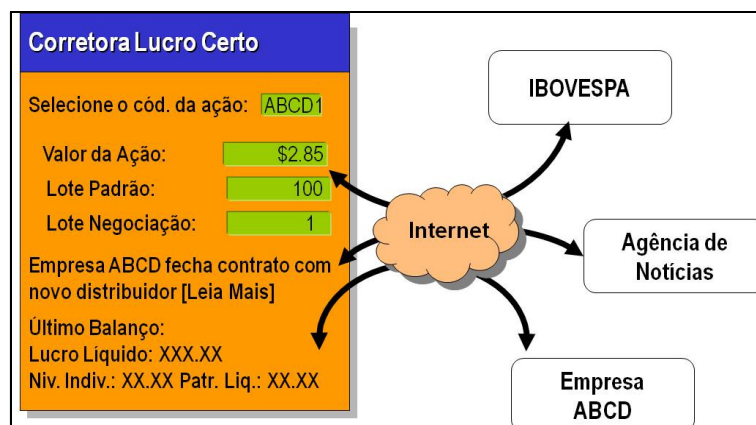
- System.Web.Services
- System.Web.Services.Description
- System.Web.Services.Discovery

## ***Como funciona a comunicação ?***

Os webservices funcionam utilizando um conjunto de protocolos padrão. Através do protocolo http, é realizado todo o transporte das informações. A troca e ativação dos serviços é feita através do SOAP, também um protocolo padrão para comunicação de objetos.

## ***Como funcionam os web services ?***

Os webservices permitem que em uma aplicação seja possível acessar informações de outros locais, como se estas informações pertencessem ao contexto da própria aplicação. Uma corretora de valores por exemplo, poderá criar um site e buscar as cotações dos papéis na bolsa de valores, o notícias sobre a empresa em uma agência de noticias e informações fiscais em um web service da própria empresa.



## ***Utilizando um web service***

Na plataforma .NET podemos utilizar um web service de duas maneiras:

- Chamada através da URL
  - o Utilizado apenas para testes
- Chamada através de uma classe proxy
  - o Utilizado em aplicações

### **Chamando um web service através de uma URL**

Para chamar um webservice através de uma URL, basta informar o endereço do webservice e apontar para o arquivo de entrada, que possui a extensão asmx.

Clicando nos métodos, é possível chama-los e conferir o seu resultado.

### **Utilizando uma classe Proxy**

A maneira mais comum de chamarmos um web service é através de uma classe proxy. O Visual Studio cria automaticamente uma classe proxy quando adicionamos uma referencia a um web service. Esta classe funciona exatamente como uma classe local no projeto, porém suas chamadas são desviadas para o webservice.

### **Adicionando uma referencia web**

Para criar uma classe proxy, basta adicionar uma nova referencia da web nas referencias do projeto. Para isto basta clicar com o botão direito sobre a pasta de referencias e selecionar a opção Add Web Reference.

### **Instanciando uma classe Proxy**

Para utilizar a classe proxy, basta criar uma instância e chamar seus métodos.

Um exemplo de chamada a uma classe proxy pode ser observado a seguir.

```
Localhost.Informacoes_Fiscais wsABC = new localhost.Informacoes_Fiscais();  
DataSet dsFiscas = wsABC.BuscaBalanco();
```

É possível alterar a localização do Web Service de forma dinâmica alterando a propriedade url behavior do serviço. Com isto, a localização do serviço passa uma chave de configuração no arquivo web config.

## ***Exercício Prático 1 – Utilizando um web service***

1. Abra o seu browser
2. Desvia seu browser para a url `HTTP://n006//link/verificacpf.aspx`
3. Acesse o método verifica CPF, informando um CPF válido
4. Abra o exercício que contém o gerenciador de contas criado nos módulos anteriores
5. Adicione uma web reference ao serviço de verificação de CPFs
6. Faça a chamada deste serviço no processo de abertura de novas contas, para que os CPFs dos novos correntistas sejam verificados.
7. Execute a aplicação e verifique o resultado

## ***Criando um web service***

O Visual Studio .NET possui um template para aplicações do tipo Web Service. A estrutura deste tipo de aplicação é a mesma de uma aplicação ASP.NET.

Para criarmos um web service precisamos criar um arquivo com a extensão `asmx`. Este arquivo possui uma diretiva de `webservice` que aponta para um arquivo `code-behind` que contém a implementação do serviço.

Um exemplo de uma diretiva de serviço pode ser observada a seguir.

```
<%@ WebService
Language="cs"
Codebehind="inf_fiscais.aspx.cs" Class="EmpresaABC.Informacoes_Fiscais" %>
```

## ***Criando métodos***

Na classe associada ao web service, devemos implementar os métodos que estarão disponíveis através da internet.

Não basta criar métodos públicos na classe para que eles estejam disponíveis através da internet, é preciso informar o atributo `WebMethod` antes da definição do método.

Um exemplo de implementação de um web method pode ser observado a seguir.

```
[WebMethod()]
public DataSet BuscaBalanco()
{
    ...
}
```

## ***Exercício Prático 2 – Criando um web service***

1. Adicione ao projeto web um novo arquivo .asmx com a definição de um web service
2. Crie um novo método chamado obterConta que devolva um objeto da classe ContaBancaria e receba um parâmetro com o número da conta
3. Implemente o método de forma que a conta seja obtida a partir do número de conta informado e devolvida como resultado do método
4. Execute o web service e teste no browser. Verifique que a instancia da conta é devolvida no formato XML, pois a classe está marcada como serializavel.