

Olá,

Criei estas apostilas a mais de 5 anos e atualizei uma série delas com alguns dados adicionais. Muitas partes desta apostila está desatualizada, mas servirá para quem quer tirar uma dúvida ou aprender sobre .Net e as outras tecnologias.

Perfil Microsoft: <https://www.mcpvirtualbusinesscard.com/VBCServer/msincic/profile>

Marcelo Sincic trabalha com informática desde 1988. Durante anos trabalhou com desenvolvimento (iniciando com Dbase III e Clipper S'87) e com redes (Novell 2.0 e Lantastic).

Hoje atua como consultor e instrutor para diversos parceiros e clientes Microsoft.

Recebeu em abril de 2009 o prêmio **Latin American MCT Awards** no MCT Summit 2009, um prêmio entregue a apenas 5 instrutores de toda a América Latina (<http://www.marcelosincic.eti.br/Blog/post/Microsoft-MCT-Awards-America-Latina.aspx>).

Recebeu em setembro de 2009 o prêmio **IT HERO** da equipe Microsoft Technet Brasil em reconhecimento a projeto desenvolvido (<http://www.marcelosincic.eti.br/Blog/post/IT-Hero-Microsoft-TechNet.aspx>). Em Novembro de 2009 recebeu novamente um premio do programa IT Hero agora na categoria de especialistas (<http://www.marcelosincic.eti.br/Blog/post/TechNet-IT-Hero-Especialista-Selecionado-o-nosso-projeto-de-OCS-2007.aspx>).

Acumula por 5 vezes certificações com o título **Charter Member**, indicando estar entre os primeiros do mundo a se certificarem profissionalmente em Windows 2008 e Windows 7.

Possui diversas certificações oficiais de TI:

- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2008
- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2005
- MCITP - Microsoft Certified IT Professional Windows Server 2008 Admin
- MCITP - Microsoft Certified IT Professional Enterprise Administrator Windows 7 Charter Member
- MCITP - Microsoft Certified IT Professional Enterprise Support Technical
- MCPD - Microsoft Certified Professional Developer: Web Applications
- MCTS - Microsoft Certified Technology Specialist: Windows 7 Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows Mobile 6. Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Active Directory Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Networking Charter Member
- MCTS - Microsoft Certified Technology Specialist: System Center Configuration Manager
- MCTS - Microsoft Certified Technology Specialist: System Center Operations Manager
- MCTS - Microsoft Certified Technology Specialist: Exchange 2007
- MCTS - Microsoft Certified Technology Specialist: Windows Sharepoint Services 3.0
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2008
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 3.5, ASP.NET Applications
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2005
- MCTS - Microsoft Certified Technology Specialist: Windows Vista
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 2.0
- MCDBA – Microsoft Certified Database Administrator (SQL Server 2000/OLAP/BI)
- MCAD – Microsoft Certified Application Developer .NET
- MCSA 2000 – Microsoft Certified System Administrator Windows 2000
- MCSA 2003 – Microsoft Certified System Administrator Windows 2003
- Microsoft Small and Medium Business Specialist
- MCP – Visual Basic e ASP
- MCT – Microsoft Certified Trainer
- SUN Java Trainer – Java Core Trainer Approved
- IBM Certified System Administrator – Lotus Domino 6.0/6.5

<b>1</b>	<b>Visão Geral do Visual Basic.NET</b>	<b>4</b>
<b>1.1</b>	<b>Conceitos Básicos</b>	<b>4</b>
1.1.1	Regras de Codificação em VB.NET	4
1.1.2	Projetos e Soluções	5
1.1.3	Classes	5
<b>1.2</b>	<b>Formulários e Controles</b>	<b>7</b>
1.2.1	Utilizando Controles	7
1.2.2	Manipulação de Controles	7
1.2.3	Debug	8
1.2.4	Compilação	9
<b>2</b>	<b>Manipulação de Variáveis</b>	<b>11</b>
<b>2.1</b>	<b>Nomeando Variáveis</b>	<b>11</b>
<b>2.2</b>	<b>Tipos de Dados do Sistema</b>	<b>11</b>
<b>2.3</b>	<b>Utilizando Variáveis CTS</b>	<b>11</b>
2.3.1	Criação e Atribuição de Valor	11
2.3.2	Operadores e Precedência	12
2.3.3	Manipulação de String	12
<b>2.4</b>	<b>Conversões Implícitas e Explícitas</b>	<b>13</b>
<b>2.5</b>	<b>Tipos Compostos</b>	<b>13</b>
<b>2.6</b>	<b>Constantes e Read Only</b>	<b>14</b>
<b>3</b>	<b>Instruções Condicionais, Laços e Desvios</b>	<b>15</b>
<b>3.1</b>	<b>Contexto de Variáveis</b>	<b>15</b>
<b>3.2</b>	<b>Comparativos</b>	<b>15</b>
3.2.1	Comando If	15
3.2.2	Comando Select Case	16
<b>3.3</b>	<b>Laços</b>	<b>16</b>
3.3.1	Comando While e Do	16
3.3.2	Comando For...Next	17
3.3.3	Comando For...Each	17
<b>3.4</b>	<b>Desvios</b>	<b>18</b>
<b>4</b>	<b>Tratamento de Erro</b>	<b>19</b>
<b>4.1</b>	<b>Erro de Execução sem Tratamento</b>	<b>19</b>
<b>4.2</b>	<b>Try...Catch...Finally</b>	<b>19</b>
4.2.1	Try...Finally	20
4.2.2	Throw	20
<b>5</b>	<b>Métodos</b>	<b>22</b>
<b>5.1</b>	<b>Definição de Métodos</b>	<b>22</b>
5.1.1	Métodos Compartilhados sem Retorno	22
5.1.2	Métodos Compartilhados com Retorno	22

---

<b>5.2</b>	<b>Recebendo Parâmetros</b>	<b>23</b>
5.2.1	Parâmetros de Entrada (ByVal)	23
5.2.2	Parâmetros Referenciais (ByRef)	24
5.2.3	Parâmetros Múltiplos	25
<b>5.3</b>	<b>Overload de Métodos</b>	<b>26</b>
<b>6</b>	<b>Arrays e Parâmetros</b>	<b>28</b>
<b>6.1</b>	<b>Definição</b>	<b>28</b>
6.1.1	Definindo Arrays	28
<b>6.2</b>	<b>Métodos Comuns</b>	<b>29</b>
<b>7</b>	<b>Classes Básicas do .NET Framework</b>	<b>30</b>
<b>7.1</b>	<b>Classe AppDomain</b>	<b>30</b>
<b>7.2</b>	<b>Classe Security</b>	<b>30</b>
<b>7.3</b>	<b>Classe IO</b>	<b>30</b>
<b>8</b>	<b>Classes e Objetos</b>	<b>32</b>
<b>8.1</b>	<b>Definição de Classes e Objetos</b>	<b>32</b>
8.1.1	Abstração e Encapsulamento	32
8.1.2	Herança	33
8.1.3	Polimorfismo	33
8.1.4	Classes Abstratas	33
8.1.5	Interfaces	34
<b>8.2</b>	<b>Criação e Instanciamento de Classes</b>	<b>34</b>
8.2.1	Propriedades e Enumeradores	35
8.2.2	Construtores	36
8.2.3	Destrutores	37
<b>8.3</b>	<b>Controle de Acessibilidade</b>	<b>37</b>
8.3.1	Métodos Compartilhados	38
<b>9</b>	<b>Herança e Polimorfismo</b>	<b>39</b>
<b>9.1</b>	<b>Classes e Métodos Protegidas</b>	<b>40</b>
<b>9.2</b>	<b>Polimorfismo</b>	<b>41</b>
<b>10</b>	<b>Namespace, Delegates e Operadores</b>	<b>43</b>
<b>10.1</b>	<b>Namespace</b>	<b>43</b>
<b>10.2</b>	<b>Delegates</b>	<b>43</b>
<b>10.3</b>	<b>Eventos</b>	<b>44</b>

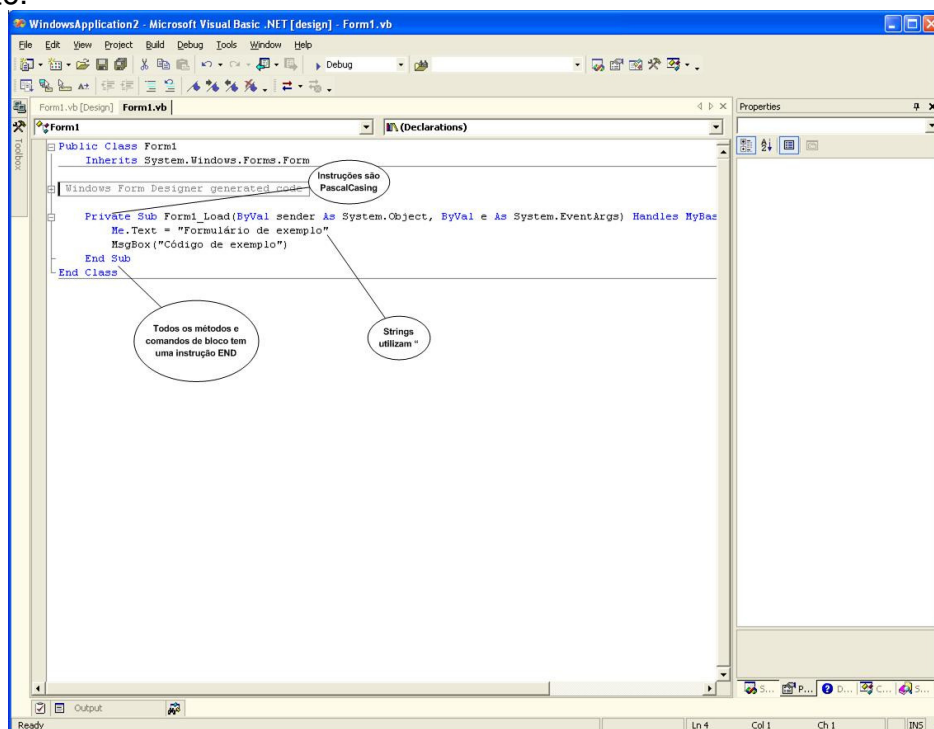
# 1 Visão Geral do Visual Basic.NET

## 1.1 Conceitos Básicos

Antes de codificar um programa dentro do Visual Studio precisamos conhecer as regras básicas de código, conceitos de projetos e classe, utilização e manipulação de propriedades dos objetos, debug e compilação.

### 1.1.1 Regras de Codificação em VB.NET

Ao utilizar duplo clique no formulário "Form1" abrimos o editor de código e inserimos apenas as das linhas dentro do bloco "Form1\_Load". O restante do código que pode ser notado faz parte do objeto formulário padrão.



Algumas das principais regras de sintaxe do VB podem ser notadas no exemplo:

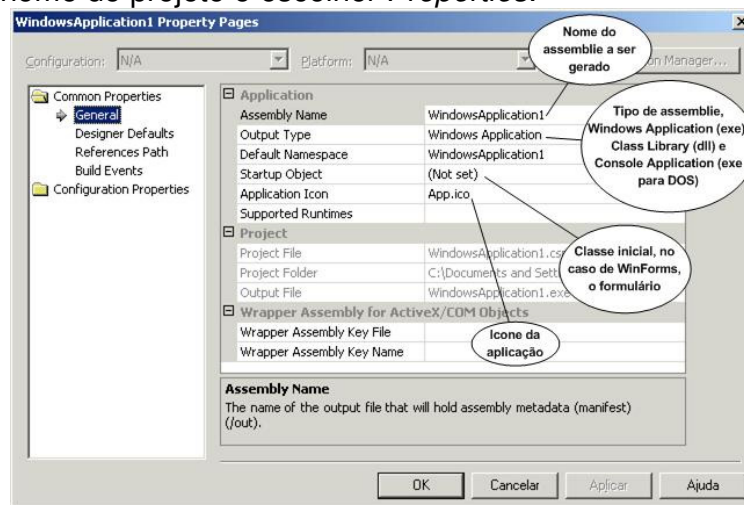
- Não existe um método inicializador dentro dos objetos do projeto. O método Main é opcional e deve constar em módulo.
- Não existe delimitador de final das linhas de instruções.
- Classes e instruções de bloco iniciam com sua declaração e terminam com *End*.
- Existem dois modelos de codificação, PascalCasing e camelCasing. Por padrão todo o framework é baseado em PascalCasing, ou seja todas as primeiras letras em maiúsculo, incluindo nas classes do CLR, como por exemplo a classe System.InteropServices. O VB.NET também tem suas instruções em PascalCasing, não sendo obrigatória a digitação exata, o VS corrige a capitulação.
- A indentação não é obrigatória para a linguagem, mas extremamente recomendável na codificação. Como pode ser visto, a indentação está na linha onde consta o *Msgbox* está no

quarto nível, sendo o nível principal o nome da aplicação, o segundo nível a classe do formulário, o terceiro nível é o método Load. No VS a indentação é automática.

- Para fazer comentários utiliza-se as duas apóstrofo “”, que podem ser usadas tanto em uma linha inteira quanto no final da linha. Ao compilar um projeto, os comentários são excluídos do assembleie.

### 1.1.2 Projetos e Soluções

No VS todas as aplicações são chamadas de projetos. Cada projeto é um assembleie, seja este um executável, dll ou aplicação web. O tipo de assembleie a ser gerado pode ser alterado por se clicar com o botão direito no nome do projeto e escolher *Properties*.



Existem algumas limitações quanto a alteração do tipo de assembleie a ser gerado. Aplicações *WinForms*, *Console* e *Library* podem ser convertidas entre elas sem problemas, mas aplicações web só podem ser convertidas entre si, no caso os projetos do tipo *Web Application* e *Web Services*. **DICA:** Diferente do Java, no .NET não há ligação entre o nome do assembleie e das classes nem no namespace (detalhes no módulo 10).

O *Startup Object* é um formulário ou o método (formulários no .NET também são classes) que contenha o método Main.

Um importante recurso provido pelo VS é trabalhar com o conceito de *Solution* ou solução. Podemos agrupar diferentes tipos de projetos em uma solução, como por exemplo, em uma única solução ter dois projetos *WinForms*, um projeto *web service* e um quarto projeto *web application*. Utilizando soluções conseguimos em uma única janela do VS aberta trabalhar com todos os projetos a que estamos envolvidos.

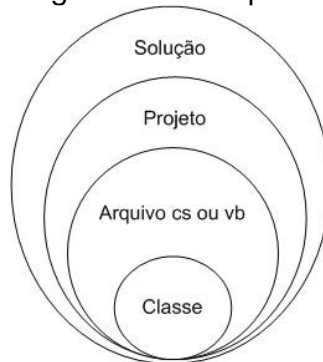
**DICA:** *Solutions* não devem ser compartilhados entre usuários, enquanto projetos sim.

Um projeto utiliza como extensão de arquivo o acrônimo da linguagem mais a palavra proj, por exemplo vbproj e cproj. Já as soluções utilizam a extensão sln e ficam sempre no diretório local do usuário, enquanto o projeto pode estar na rede ou no servidor web quando a aplicação é do tipo *web application*.

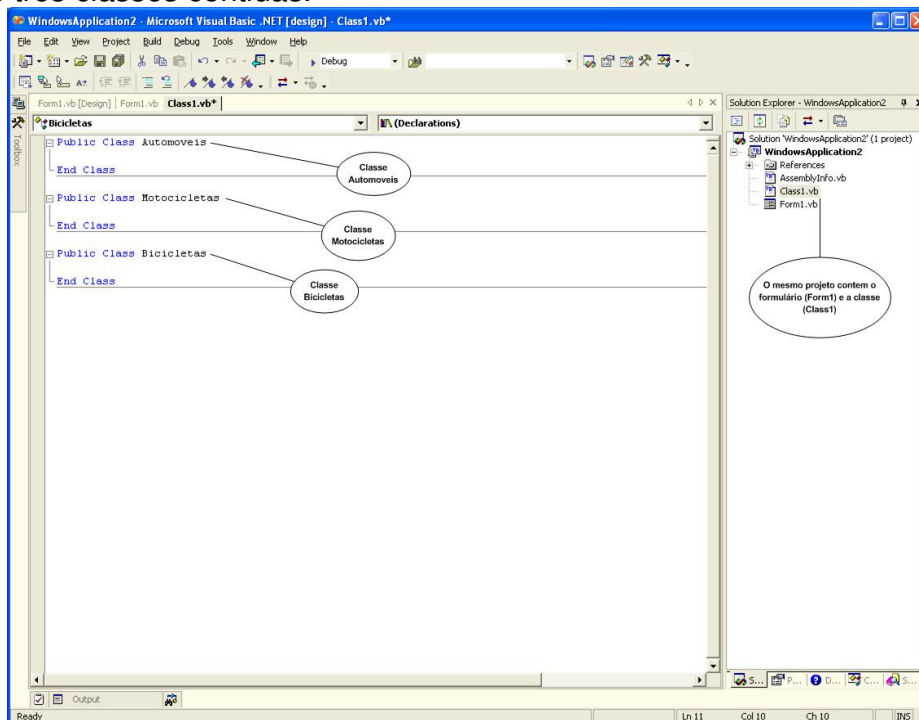
### 1.1.3 Classes

Não entraremos em detalhes agora sobre recursos das classes, mas precisamos entender a estrutura de um projeto, e para isso é essencial entender o conceito das classes.

Classes são objetos que podem ou não se tornar componentes. Damos o nome componente aos assemblies, e um mesmo assembly pode conter diversas classes. Da mesma maneira, o arquivo físico no disco que contém as classes, extensão *cs* para C# e *vb* para Visual Basic, podendo conter múltiplas classes internamente. Veja o diagrama abaixo para entender melhor este conceito.



Veja no projeto abaixo um exemplo de uma solução, um projeto, dois arquivos de classes VB e em um único arquivo três classes contidas.



Apesar de estarem dentro do arquivo *Class1* as três classes são totalmente independentes ao serem compiladas. Ou seja, não importa o arquivo físico onde as classes estão, uma vez que na compilação não existem arquivos físicos, apenas classes. Imagine que no assembly compilado estará apenas as classes *Form1*, *Automóveis*, *Motocicletas* e *Bicicletas* dentro do componente *WindowsApplication2*.

**DICA:** Um arquivo pode conter múltiplas classes, mas uma classe não pode ser em múltiplos arquivos.


























Uma classe sempre utiliza um escopo, neste caso *public* (detalhes no módulo 6), a palavra chave *class* e o nome definido. Todas as classes iniciam e terminam com os delimitadores de chave.

## 1.2 Formulários e Controles

### 1.2.1 Utilizando Controles

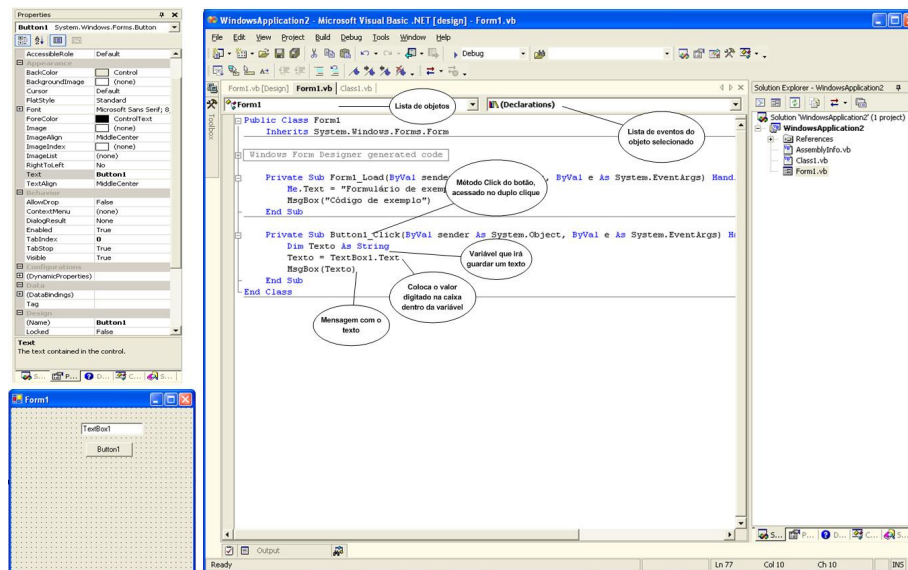
Controles são os objetos gráficos utilizados em formulários.

Ao utilizar formulários, seja do tipo windows ou web, podemos arrastar os controles da barra de ferramentas (*toolbox*). Segue abaixo uma lista dos principais controles utilizados em aplicações WinForms.




Ícone	Função
 StatusBar	A barra de rodapé fixa utilizada nos aplicativos como Office, Explorer e outros para mostrar estado de botões, data e hora, e qualquer outra dado utilizando painéis
 TabControl	Guias como as de propriedades permitindo múltiplas janelas em um único espaço
 TextBox	Caixa para digitação, possui a propriedade multiline para maiores.
 Timer	Em intervalos freqüentes executa o método timer
 ToolBar	Barra de botões 3D como a utilizada nos aplicativos do Windows.
 ToolTip	Caixas automáticas de ajuda
 TreeView	Lista de textos, imagens e qualquer outro tipo de informação, utilizada no Windows Explorer para demonstrar a lista de diretórios
 Button	Botão de comando com texto ou imagem de fundo
 CheckBox	Caixa de ligado e desligado para múltiplas opções simultâneas
 CheckedListBox	Lista com múltiplos checkbox em um único espaço
 ComboBox	Lista de opções com única escolha
 ContextMenu	Menu de botão direito
 CrystalReportViewer	Visualizador de relatórios criados com o Crystal Report embutido no VS2003
 DataGrid	Grid de dados
 DateTimePicker	Caixa de texto com botão para mostrar calendário dinâmico
 Label	Caixa de texto fixa. O LinkLabel permite chamar páginas Internet executando o browser
 LinkLabel	
 ListBox	Lista com diversos itens, podendo ser configurada para múltiplas opções
 MonthCalendar	Calendário aberto com navegação, utilizado no Windows
 NumericUpDown	Numérico com valores máximo e mínimo com botões para alteração
 PictureBox	Imagem com controle de click
 ProgressBar	Barra de progresso, utilizada no Internet Explorer para indicar progresso
 RadioButton	Botões de opção para única escolha
 NotifyIcon	Ícone no tray icon do Windows
 MainMenu	Menu para formulários

### 1.2.2 Manipulação de Controles

Para manipular estes controle em tempo de design (tela de edição gráfica do VS) utilizamos a janela de propriedades (*properties*). Mas quando precisamos manipular as propriedades em tempo de execução (*runtime*) utilizamos o nome do objeto e acessamos suas propriedades. O exemplo abaixo demonstra este principio, onde foi colocado um *label*, um *textbox* e um botão e o objetivo é digitar um texto no textbox e ao clicar no botão o texto ser colocado no label.



Neste caso acessamos o método Click por ser o método padrão, bastando utilizar duplo clique. Outros métodos de um objeto podem ser acessados por escolher este na lista de objetos e os eventos na lista ao lado.

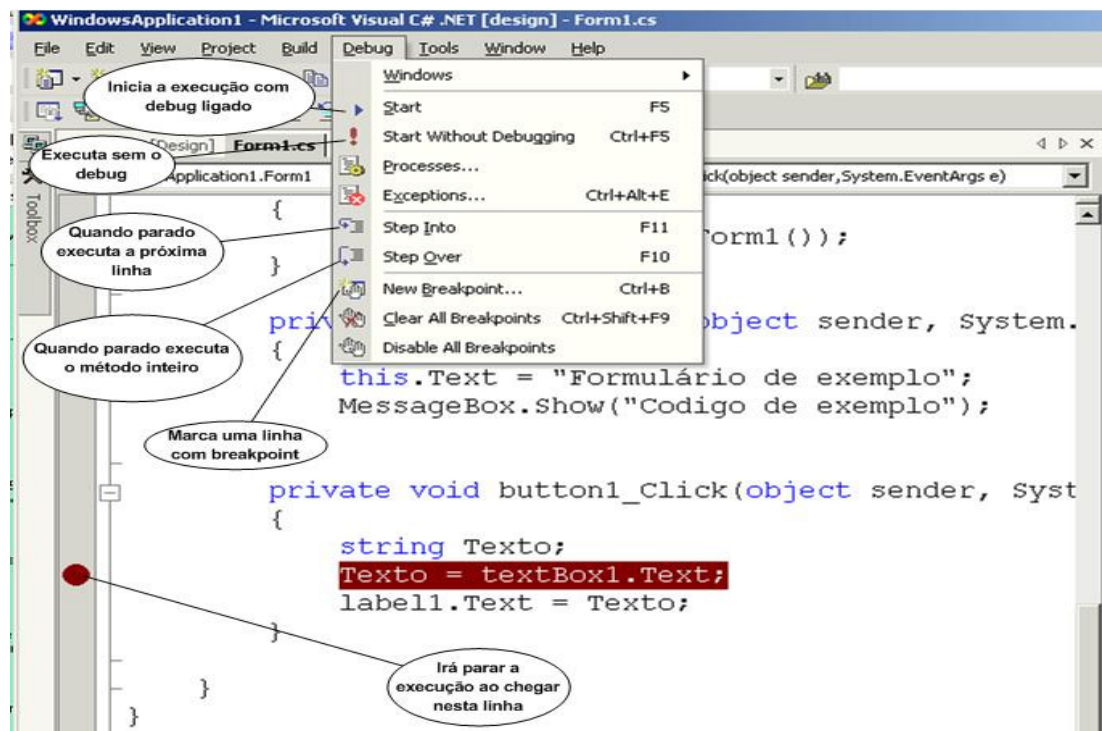
Ao digitar na janela de código o nome do objeto automaticamente o VS mostra uma lista com todos os eventos, métodos e propriedades que o controle possui. O símbolo  indica propriedade,  indicam os métodos e  os eventos.

### 1.2.3 Debug

Os recursos de debug do VS são muito úteis e permitem ver em tempo real a linha de código que está sendo executada, permitindo com uma variedade de janelas auxiliares verificar valores de variáveis (watch), valores atuais (local) e outras que ficam ativas quando executamos o projeto (tecla F5).

Para poder fazer o debug de um programa temos duas diferentes formas. A primeira é utilizar a janela watch para definir um *break* que consiste em colocar o nome da variável ou propriedade que está debugando e escolher entre parar a execução quando mudar de valor, ao ter um determinado valor ou em determinado local de alteração. É um modo muito útil quando temos problemas com uma variável e não sabemos qual é o local ou situação onde este valor está sendo alterado.

A segunda forma de fazer debug é marcar uma linha de *breakpoint*, ou ponto de parada, clicando-se na linha e utilizando o menu debug ou clicando-se na barra lateral de configuração, como o exemplo abaixo demonstra.



Os recursos *Step Into* e *Step Over* são especialmente importantes por permitirem passar o teste a linha seguinte ou pular aquele método até o breakpoint seguinte.

## 1.2.4 Compilação

Compilar um programa feito em linguagens como Visual Basic 6, C += 1, Delphi e algumas outras linguagens geramos um código que para ser executado não necessita de um framework como o .NET. A este modelo chamamos de código nativo.

Como o .NET precisa do framework para poder funcionar, não existe código nativo de máquina, mas sim o IL como citado no módulo 1. Veja abaixo o método click do botão do formulário criado anteriormente em IL:

```
.method private hidebysig instance void button1_Click(object sender,
class [mscorlib]System.EventArgs e) cil managed
{
    ' Code size      25 (0x19)
    .maxstack 2
    .locals init ([0] string Texto)
    IL_0000: ldarg.0
    IL_0001: ldfld    class [System.Windows.Forms]System.Windows.Forms.TextBox WindowsApplication1.Form1::textBox1
    IL_0006: callvirt instance string [System.Windows.Forms]System.Windows.Forms.Control::get_Text()
    IL_000b: stloc.0
    IL_000c: ldarg.0
    IL_000d: ldfld    class [System.Windows.Forms]System.Windows.Forms.Label WindowsApplication1.Form1::label1
    IL_0012: ldloc.0
    IL_0013: callvirt instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
    IL_0018: ret
} ' end of method Form1::button1_Click
```

Como podemos notar, o código não é o mesmo que escrevemos originalmente, mas ele possui as classes completas dos controles utilizados, como por exemplo a linha

`[System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)`, onde é referenciado o textbox e utilizando o `set` para alterar a propriedade texto com uma string.

Para que um programa escrito em linguagens do .NET se transformem em IL é necessário um compilador específico. Por exemplo, o compilador do C# se chama `csc.exe` e o compilador do VB é o `vbc.exe`. Utilizando o compilador do C# para criar a aplicação utilizada até o momento utilizamos a seguinte linha de comando:

```
vbc /target:exe /out:MeuExecutavel.exe Class1.cs Form1.cs
```

Nesta linha definimos o tipo de assembleia, neste caso executável, o nome do assembleia e os arquivos físicos de classes que devem ser incluídos.

*DICA: Para poder usar linhas de comando do .NET utilizamos o “VS .NET 2003 Command Prompt” e não o console normal de comando do Windows.*

Outro importante aplicativo utilizado na linha de comando é o `ngen.exe` para evitar o IL. Caso utilize este aplicativo ao invés de criar um IL teremos um assembleia já interpretado para .NET, o que não elimina a necessidade do framework para acessar os objetos, mas evita que o “Code Manager” e o “MSIL to Native Compiler” tenham que ser utilizados nas execuções deste programa.

Outros aplicativos como o `sn.exe`, `certmgr.exe`, `gacutil.exe`, etc. são tratados nas apostilas de framework avançado e enterprise services.

## 2 Manipulação de Variáveis

### 2.1 Nomeando Variáveis

Algumas regras básicas devem ser seguidas ao dar nome a variáveis:

- Não devem começar com números, sempre com letras
- Evite utilizar “\_”
- Utilize PascalCasing
- Evite abreviações ou nomes não relacionados
- Não utilize palavras reservadas.

### 2.2 Tipos de Dados do Sistema

O .NET Framework é quem cria, manipula e gerencia variável. Este processo é interessante por permitir que todas as linguagens tenham os mesmos tipos de dados, sendo este o motivo de interoperabilidade sem problemas de compatibilidade. Este conjunto de variáveis é chamado de CTS (Common Type Safe), sendo controlado pelo componente *Type Checker* do CLR. Os principais tipos de dados que o CTS possui são referenciados pelo nome deles ou então pelos alias (apelidos) padronizados em todas as linguagens e banco de dados. Segue a lista dos tipos e seus alias utilizados no VB.NET:

Alias	Nome de Sistema
Sbyte	System.Sbyte
Byte	System.Byte
Short	System.Int16
Integer	System.Int32
Long	System.Int64
Char	System.Char
Single	System.Single
Double	System.Double
Boolean	System.Boolean
Decimal	System.Decimal
String	System.String

### 2.3 Utilizando Variáveis CTS

O CTS suporta variáveis de sistema (citadas anteriormente) e objetos, como por exemplo, formulários, classes, controles, etc.

A forma de transferência e assimilação de valores pode ser feita pelo modo valor (ByVal) ou pelo modo referencia (ByRef). Todos as variáveis criadas de objetos são do tipo referencia, uma vez que o tipo de dados é parte do framework e o que se cria é apenas um apontador para o objeto. Por exemplo, ao criar uma variável do tipo Dataset não se está criando um novo dataset. Para isso precisa-se utilizar a palavra chave *New*.

#### 2.3.1 Criação e Atribuição de Valor

Ao criar uma variável do tipo CTS a sintaxe básica é formada pelo tipo da variável, pode ser o alias, e o nome. Opcionalmente pode-se atribuir o valor inicial sem a necessidade de outra linha. Compare os dois exemplos abaixo:

*Dim Idade As Integer* 'Variável numérica de nome idade sem valor  
*Idade = 19* 'Atribuído um valor numérico inteiro  
*Dim Nome As String = "Fulano"* 'Variável texto Nome com valor já definido

Utilizamos aqui variáveis CTS e seus respectivos alias, mas poderíamos ter usado os nomes de sistema:

*Dim Idade As System.Int16* 'Variável numérica de nome idade sem valor  
*Idade = 19* 'Atribuído um valor numérico inteiro  
*Dim Nome As System.String = "Fulano"* 'Variável texto Nome com valor já definido

Para alteração de valores da variável utilizamos o sinal "=" mesmo quando a alteração é somar, dividir e outros, como o exemplificado abaixo:

*Dim Contador As Integer = 0* 'Criação com valor 0  
*Contador += 1* 'Soma um  
*Contador -= 1* 'Subtrai um  
*Contador /= 2* 'Divide por 2

Segue a lista com todos os operadores, função e exemplo:

Operador	Função	Exemplo
=	Altera valor de variáveis CTS e compara igualdade	Nome = "Fulano"
<>	Nome diferente de Fulano	Nome <> "Fulano"
> e >=	Maior (31 para frente) e maior ou igual (inclui o 30)	Idade > 30 Idade >= 30
< e <=	Menor (29 para baixo) e menor ou igual (inclui o 30)	Idade < 30 Idade <= 30
is	Compara igualdade entre objetos	If TypeOf Button1 Is Button Then
And	Significa que as comparações devem ser todas verdadeiras	Idade = 30 And Nome = "Fulano"
Or	Significa apenas uma das comparações precisa ser verdadeira	Idade = 30 Or Nome = "Fulano"
iif	Condicional true/false	iif(Nome=(x-y), "OK", "Erro")
+= 1	Soma um a variável numérica	Idade += 1
Mod	Divide um numero retornando o resto	Idade Mod 3 = 0.50

## 2.3.2 Operadores e Precedência

Assim como em cálculos matemáticos, existe uma precedência em processamento de variáveis e valores. Por exemplo, na equação  $1+2*3/4$  o resultado será dois e meio uma vez que multiplicação e divisão são executadas antes da adição e subtração.

Para servir de referencia siga a seguinte regra: \*, /, %, +, -, And e Or.

Como alternativa para controle de precedência utilize parênteses. Por exemplo ao invés de escrever *A Or B And C* escreva *(A Or B) And C*. Vamos entender a diferença.

No primeiro exemplo B e C tem que ser igual e A não faz diferença. No segundo exemplo A e B não fazem diferença, qualquer um deles pode ser combinado com C. Aqui fica clara a diferença feita pelo controle de precedência.

## 2.3.3 Manipulação de String

Variáveis do tipo string são especiais por permitirem uma série de operações especiais. A tabela abaixo demonstra o efeito de cada um dos métodos string:

Operação	Valor da Variável	Resultado
<i>IndexOf("a")</i>	Produtividade	10
<i>Insert(3, "xxx")</i>	Maria	Marixxxa
<i>LastIndexOf(s, "a")</i>	Variavel	4
<i>Length</i>	Maria	5
<i>PadLeft(5, "0")</i>	20	00020
<i>PadRight(5, "0")</i>	20	20000
<i>Remove("ar")</i>	Maria	Mia
<i>Replace("ari", "vvv")</i>	Maria	Mvvva
<i>Split("")</i>	MariaJoãoJoaquim	Um em cada posição do array

<b>Substring(2,4)</b>	Produtividade	Odut
<b>ToLower</b>	Produtividade	Produtividade
<b>Tostring</b>	Inteiro 60	String 60
<b>ToUpper</b>	Produtividade	PRODUTIVIDADE
<b>Trim</b>	Produtividade	Produtividade

Variáveis string são tratadas como array, portanto ao utilizar o numero dois não estamos chamando a segunda letra mas sim a terceira, pois numerações de conjuntos iniciam baseados no zero e não no numero um.

## 2.4 Conversões Implícitas e Explícitas

Conversões implícitas são feitas sem a necessidade de transformar um valor em outro. Isto só pode ser feito quando duas variáveis são do mesmo tipo principal, como por exemplo, dois valores numéricos ou valores string. Veja os exemplos abaixo:

```
Dim Numero As Integer
Dim Convertido As Long= Numero 'Funciona pois os dois são números
Dim Convertido As String= Numero 'Funciona por cast automático
Numero = "888" 'Funciona por cast automático
```

No primeiro exemplo a variável long é numérica e maior que a variável inteira, portanto não é necessário converter.

Para conversões explícitas no VB utilize as funções incorporadas a linguagem:

```
Dim Numero As Integer
Numero = Cint("888")
Dim Nome As String
Nome = Numero.ToString()
```

As funções de conversão no VB são sempre precedidas por um "c" e o acrônimo do dado desejado, ou seja, clng, cint, cbol, cdbl, etc.


Para conversão de um numero para Char utiliza-se o Chr(x), e de um caractere para ASCII a instrução Asc('x').

## 2.5 Tipos Compostos

Existem dois tipos compostos no .NET Framework. São os valores do tipo *Enum* e *Structure*.

Os valores do tipo *Enum* servem fornecer uma lista de valores possíveis, e permite que se utilize um nome ao invés de um valor numérico. Como exemplo imagine uma aplicação onde temos que definir o sexo das pessoas envolvidas em um processo de seleção. O exemplo abaixo demonstra como este código seria:


```
Enum Sexo As Integer
    Masculino = 1
    Feminino = 2
    Ignorado = 3
End Enum
Dim NomeFuncionario As String= "Maria"
Dim SexoFuncionario as Integer = Sexo.Feminino
Msgbox(SexoFuncionario.ToString()) 'Resultado = 2
```

Como notado, é muito mais fácil digitar Sexo e depois o tipo do que decorar que o numero um é masculino e o numero 2 é feminino. A praticidade do enum é reconhecida pois o próprio .NET utiliza enum em muitas de suas classes. Sabemos da existência de um *Enum* pelo símbolo  na lista de um objeto.

Por outro lado, enquanto o enum cria uma lista de possibilidades com valores numéricos ou caracteres fixos, o *Structure* monta uma variável composta por diversos dados. Por exemplo, criar

um *Structure* chamado funcionário que contenha nome, idade, telefone e endereço. Veja o exemplo de enum agora combinado com *Structure*:

```
Structure Funcionario
    Dim Nome As String
    Dim Idade As String
    Dim Telefone As String
    Dim Endereco As String
End Structure
Enum Sexo As Integer
    Masculino = 1
    Feminino = 2
    Ignorado = 3
End Enum
Funcionario Maria
Maria.Nome = "Maria"
Maria.Idade = 20
Maria.Telefone = 89897676
Maria.Sexo = Sexo.Feminino
Msgbox(Maria.Sexo.ToString()) 'Resultado = 2
```

Este exemplo demonstrou muito bem como um *Structure* ajuda, pois ele é utilizado como uma única variável, neste exemplo chamamos de funcionário, e criamos uma variável baseada no *Structure*, no exemplo Maria, e esta variável contém os atributos que um funcionário deve ter. A vantagem de um *Structure* é montar uma estrutura pronta, uma vez que evita a um programador esquecer ou deixar de informar determinado valor para um funcionário, pois o funcionário já contém automaticamente os quatro atributos, mesmo que não informados. O sinal que indica um *Structure* é .

## 2.6 Constantes e Read Only

Constantes são variáveis com valores fixos. Por exemplo, o PI matemático é um constante de (aproximadamente) 3,141516.

Podemos criar constantes para certos dados muito utilizados em uma aplicação, como o exemplo abaixo:

```
Const Aplicação As String = "Curso"
Const Cidade As String = "Taquaritinga"
Msgbox(Aplicacao + " --> " + Cidade)
```

Notamos que as variáveis aplicação e cidade não podem ser alteradas e são utilizadas como qualquer outra variável do sistema.

Diferentes das constantes podemos ter variáveis que são alteradas em uma classe, mas não podem ser alteradas fora da classe em que foram criadas. Para isso utilizamos a instrução *readonly* após a definição da variável, como o exemplo a seguir:

```
public readonly Aplicacao As String
public readonly Cidade As String
Aplicacao = "Curso"
Cidade = "Taquaritinga"
```

**DICA:** Variáveis *read only* só podem ser alteradas no construtor da classe em que foram criadas.

## 3 Instruções Condicionais, Laços e Desvios

### 3.1 Contexto de Variáveis

Ao utilizarmos condicionais, laços e desvios precisamos juntar o conceito de bloco com criação de variáveis.

Aprendemos anteriormente que blocos são criados utilizando-se as chaves, como os exemplos anteriores de *Enum*, *Structure* e classes. Variáveis criadas dentro de um bloco só podem ser utilizadas dentro do mesmo bloco. Este princípio é chamado de contexto. O exemplo abaixo demonstra uma variável dentro e outra fora do contexto:

```
Dim Idade As Integer

If Idade = 30 Then
    Dim Peso As Integer
    Idade = 30
    Peso = 60
End If
Msgbox(Idade.ToString())      'Funciona normalmente
Msgbox(Peso.ToString())      'Erro de variável não definida
```

Notamos que dentro do bloco a variável *Idade* pode ser utilizada normalmente, uma vez que ela foi criada antes do bloco. Já a variável *Peso* que foi criada dentro do bloco só existe dentro do bloco, portanto no *Msgbox* ocorreu um erro porque a variável já não existia.

### 3.2 Comparativos

#### 3.2.1 Comando If

A sintaxe básica do comando *if* é:

```
If condição Then      'condição verdadeira
    comandos
Else                  'condição falsa
    comandos
End If
```

Como primeiro exemplo do comando imaginemos uma aplicação onde validamos a idade que o usuário digita:

```
Dim Idade As Integer
Idade = Cint(textBox1.Text)
If Idade <= 0 Or Idade >= 100 Then
    MsgBox("Idade Incorreta. Digite Novamente")
    textBox1.SetFocus()
Else
    MsgBox("Idade aceita. Obrigado")
End If
```

No exemplo utilizamos *Or* para indicar que se a idade digitada for menor ou igual a 0 ou então maior ou igual a 100 a mensagem de erro aparece e obrigamos a digitar novamente. Caso contrário, ou seja, se estiver entre 1 e 99 mostramos uma mensagem de agradecimento.

Ampliando o nosso exemplo pode validar também o nome da pessoa:

```
Dim Idade As Integer
Idade = Cint(textBox1.Text)
If Idade <= 0 Or Idade >= 100 Then
    MsgBox("Idade Incorreta. Digite Novamente")
    textBox1.SetFocus()
```

```

Else
    Dim Nome As String
    Nome = textBox2.Text
    If Nome.Length < 5 Or Nome.Length >30 Then
        MsgBox("Nome Incorreto. Digite Novamente")
        textBox2.SetFocus()
    Else
        MsgBox("Idade aceita. Obrigado")
    End If
End If

```

Note que caso a idade esteja correta agora o *Else* possui um bloco, e dentro deste bloco lemos o nome escrito no textbox e verificamos o tamanho do nome. Se o nome for menor que 5 ou maior do que 30 caracteres, voltamos ao textbox para que ele redigite o nome. Caso esteja correto entra no segundo *Else* que mostra a mensagem de dados aceitos.

*DICA: O comando If encadeado só deve ser utilizado quando as condições são diferentes. Não se utiliza quando a condição é com a mesma variável. Nestes casos utiliza-se o Select Case.*

### 3.2.2 Comando Select Case

A sintaxe básica do comando *Select Case* é:

```

Select Case Variável
    Case 1
        instrução
        break
    Case 2 Or 3 Or 4
        instrução
        break
    Case Else
        instrução
        break
End Select

```

Para utilização do *Select Case* utiliza-se um bloco definido e uma condição única.

Cada diferente valor assumido por esta condição é verificada em uma instrução *Case*, seguida do valor comparado e operadores lógicos. A instrução *Case Else* sinaliza a instrução caso nenhum dos *Case* anteriores retorne verdadeiro. No exemplo, se o valor for acima de cinco entrará as instruções do *Case Else* já que os *Case* não retornaram verdadeiro.

Veja abaixo um exemplo comparando o tamanho do texto digitado:

```

Texto = textBox1.Text           'Atribui a variável
Select Case Texto.Length       'Compara o tamanho da string
    Case 10 Or 11 Or 12       'Os três valores ocorrem como única condição
        MsgBox("Acima de 10 até 12")
    Case 13:
        MsgBox("Valor é 13")
    Case Else                 'Caso o valor esteja diferente de 10 a 13
        MsgBox("Não encontrei...")
        break
End Select

```

## 3.3 Laços

### 3.3.1 Comando While e Do

Estes dois comandos possuem a sintaxe e funcionalidade similar, sendo o momento de comparação a principal diferença entre eles. A sintaxe básica da instrução *While* é:

```
Do While condição
    instruções
Loop
```

A sintaxe básica da instrução *do* é:

```
Do
    instruções
Loop While condição
```

Como pode ser notado entre os dois comando a diferença é que no *while* fazemos a comparação de condição na entrada do bloco, portanto pode acontecer de nem sempre ser executado, enquanto o *do* processa a primeira execução e a comparação é validada para continuação do laço.

Por exemplo, veja os dois códigos abaixo:

```
Dim Contador As Integer = 1      'Variável inicia em 1
Do While Contador > 10          'Este código nunca irá rodar
    Console.WriteLine("Rodei...while")
    Contador += 1
Loop
'Executa agora o do com a mesma condição
Do                               'Este código roda a primeira vez pois a comparação só está no final
    Console.WriteLine("Rodei...do")
    Contador += 1
Loop While Contador < 10
```

### 3.3.2 Comando For...Next

É possível utilizar o comando *While* e *Do* para processamento de laços contínuos, mas neste caso desperdiçamos algumas linhas de código para definir a variável e incrementa-la. Já o comando *For* permite que façamos laços seqüências com poucas linhas. Sua sintaxe básica é:

```
For variável = x To y Step incremento
    instruções
Next
```

Veja o exemplo anterior utilizando a instrução *For*:

```
Dim Contador As Integer
For Contador = 1 To 10 Step 1
    Console.WriteLine("Rodei..{0}", Contador)
Next
```

No exemplo criamos a variavel contador antes da instrução do tipo inteiro. Na segunda linha informamos que o laço irá começar em 1 e terminar em 10, de 1 a 1. O *Step* não é obrigatório quando o incremento é apenas um.

### 3.3.3 Comando For...Each

O comando *For* possui a limitação de trabalhar com uma condição pré-fixada em código. Isto algumas vezes limita as possibilidades de contar objetos ou coleções onde não temos o numero exato de ocorrências. Quanto queremos comparar coleções utilizamos o *For...Each*, uma vez que ele faz o laço não por comparação de valores mas sim por quantidade de itens em um conjunto, qualquer que seja este. A sintaxe básica do *For...Each* é:

```
For Each Variavel in Conjunto
    instruções
Next
```

Para exemplificar a diferença entre o *For* e o *For...Each* imagine uma coleção com 10 nomes criados abaixo:

```
Dim Nomes(10) As String
Dim Laco As Integer
```

```
For Laco = 1 To Nomes.Length
    Dim NomeCorrente As String
    NomeCorrente = Nomes(Laco)
    Console.WriteLine(NomeCorrente)
Next
Dim NomeAtual As String
For Each NomeAtual in Nomes
    Console.WriteLine(NomeAtual)
Next
```

Note que o primeiro *For* necessitamos utilizar um contador, comparar até quando este contador é válido e utilizar o item array, o numero utilizado como índice.

No segundo, utilizando o *For...Each*, para cada string que o conjunto de strings *Nomes* contem é atribuído uma variável *NomeAtual* e esta é impressa sem a necessidade do índice.

Algumas considerações sobre *For...Each* é que o tipo tem que ser o mesmo do conjunto, para cada índice do conjunto é alimentado uma variável temporária que aponta para o valor e automaticamente termina ao final da coleção. Isto pode ser visto no exemplo, pois *Nomes* é um conjunto de dez strings e a cada diferente posição a variável *NomeAtual* ganhava o valor da seqüência. Portanto, a variável *NomeAtual* alterou de valor dez vezes.

### 3.4 Desvios

Apesar de ser uma linguagem estruturada o VB possui algumas características de linguagens estruturais como pulos de código em caso de códigos condicionais particionados. As instruções que permitem desvios são *Goto* e *Exit*.

A sintaxe básica das instruções de desvio são:

```
Private Sub Teste()
    Dim Valor As Integer = Cin(textBox1.Text)
    If Valor = 10 Then
        Goto Escape1
    Else
        Goto Final
    End If
Exit Sub

Escape1:
    Console.WriteLine("Entrei no escape")
    Goto Final

Final:
    Console.WriteLine("Estou saindo")
End Sub
```

No exemplo se o valor digitado for 10 desviamos a execução para os código dentro do bloco *Escape1* que redireciona para o bloco *Final*.

Este modelo de programação não é muito utilizado por ser desestruturado, mas em certas situações podemos utilizar para controlar dentro de uma condicional situações específicas. Por exemplo, em uma instrução *Select Case* quando existem diversas condições e cada condição possui diversas linhas de código, a visualização das condições e instruções pode ficar prejudicada pelo tamanho.

Neste caso o *Goto* melhora a leitura do *Select Case* sem criar métodos adicionais no sistema.

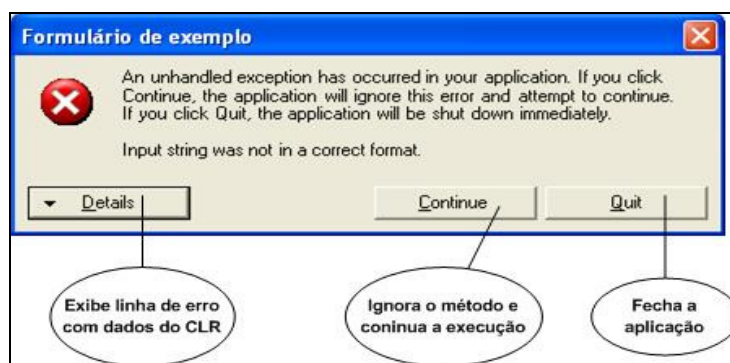
As instruções *Exit For*, *Exit Sub*, *Exit Function* e *Exit Do* são utilizadas dentro de seus respectivos blocos para indicar que deseja fechar o método ou o laço.

## 4 Tratamento de Erro

### 4.1 Erro de Execução sem Tratamento

Quando executamos um bloco de códigos precisamos tomar o cuidado de não deixar o sistema parar a execução com código de erro padrão do CLR. Este tipo de mensagem não é facilmente entendida pelo usuário, além de passar a impressão de um código mal feito e instável.

Em caso de erros no CLR é mostrada uma mensagem de tratamento permitindo que o sistema continue a ser utilizada, mas aquele método será desconsiderado, ou então pode-se escolher fechar a aplicação.



### 4.2 Try...Catch...Finally

Para controlar e melhorar utilizamos as instruções *Try*, *Catch* e *Finally*. Primeiro vamos explorar o *Try...Catch* que possui a sintaxe abaixo:

```
Try
    instruções
Catch Variavel As Exception
    instruções
End Try
```

Como exemplo, podemos fazer o teste alterando uma variável para ocorrer um erro e trata-lo, conforme o exemplo abaixo:

```
Dim Numero As Integer = 12345
Try
    Numero *= 888888
    Console.WriteLine("Numero alterado.")
Catch Erro As Exception
    MsgBox("Ocorreu um erro na aplicação. " + Erro.Message)
End Try
```

No código as duas linhas dentro do bloco delimitado pelo *Try* são executadas e em caso de erro dispara uma mensagem com o texto do erro ocorrido.

Apesar de ser permitido utilizar um bloco de tratamento dentro de outro, o VS não permite pois o tratamento de erro fica retrogrado, executando todos os que estejam dentro. Por outro lado a instrução *Catch* pode conter diversas condições, mas para entender precisamos primeiro conhecer as classes *Exception*.

Em algumas linguagens, como por exemplo o VB6, os erros eram identificados pelo numero. Isto podia ser um problema porque alguns erros eram específicos de banco dados, acesso, segurança,

disco e outros, e para descobrir precisamos comparar o numero e ler a string de retorno para tomar a ação corretiva.

Em .NET esta problema não acontece porque cada tipo de erro possui seu próprio conjunto, ou coleção, de atributos. Por exemplo, *OverflowException*, *SQLDataAccessException*, *RankException*, *SyntaxErrorException*, etc.

Utilizando este modelo de erros do .NET, imagine o exemplo abaixo:

```
Dim Numero As Integer = 12345
Try
    Numero *= 888888
    Console.WriteLine("Numero alterado.")
Catch Estouro As OverflowException
    MsgBox("Valor maior que o possivel.")
Catch Erro as Exception
    MsgBox("Ocorreu um erro na aplicação. " + Erro.Message)
End Try
```

Note que o *Exception* anterior não foi retirado, assim podemos garantir que se o erro for estouro irá ocorrer o primeiro *Catch*, mas caso ocorra um erro diferente de estouro o código seguinte, genérico, é executado e mostra a mensagem do CLR.

#### 4.2.1 Try...Finally

O *Finally* tem uma função agregada as outras instruções de tratamento de erro, sendo executada com ou sem erro. O exemplo anterior atualizado seria:

```
Dim Numero As Integer = 12345
Try
    Numero *= 888888
    Console.WriteLine("Numero alterado.")
Catch Estouro As OverflowException
    MsgBox("Valor maior que o possivel.")
Catch Erro as Exception
    MsgBox("Ocorreu um erro na aplicação. " + Erro.Message)
Finally
    MsgBox(Numero.ToString())
End Try
```

O código encontrado no *Finally* irá mostrar uma mensagem com o numero resultado da operação, tendo sido multiplicado ou não.

#### 4.2.2 Throw

Ao utilizarmos códigos em classes e termos um formulário ou página web executando este código não podemos deixar um erro ocorrido na classe parar o sistema sem avisar a fonte, quem chamou a classe. A este processo chamamos de *host* o sistema que utiliza a classe, e a classe chamamos de *objeto*.

Nestes casos precisamos que o erro ocorrido na classe seja enviado para o host, e este é que precisará mostrar o erro conforme as regras do software. Para isso utilizamos a instrução *Throw*. A sintaxe desta instrução é:

```
Throw New Exception("Mensagem de Erro")
```

Para exemplificar o processo, veja o exemplo abaixo de uma classe:

```
Public Class Cálculos
    Public Sub Multiplica(N1 As Integer, N2 As Integer)
        Dim Resultado As Integer
        Try
            Resultado = N1 * N2
        Catch Erro As Exception
            Throw New Exception("A multiplicação não foi realizada.")
        End Try
    End Sub
End Class
```

```
        End Try
    End Sub
End Class
```

Agora veja o exemplo de um programa que utilize a classe:

```
Public Class Teste
    Public Sub Main()
        Dim objMatematico As Calculos = new Calculos()
        Try
            objMatematico.Multiplica(333,444)
        Catch Erro As Exception
            MsgBox(Erro.Message)
        End Try
    End Sub
End Class
```

Ao chamar o código da classe de cálculos garantimos que os erros ocorridos serão lançados até quem a utilizou. Isto garante que o erro seja mostrado e tratado na aplicação, por exemplo, uma aplicação pode gravar em arquivo, outra enviar email e assim por diante. Se a classe executasse o código de erro internamente seria impraticável os diferentes tipos de log de erro.

## 5 Métodos

### 5.1 Definição de Métodos

Métodos são porções de códigos que podem ser executados por outros códigos. São comuns em qualquer linguagem, disponibilizando aproveitamento de funcionalidades entre diferentes partes de um mesmo código, além de interagirem com objetos. Alguns exemplos de métodos é o click do botão, o load do formulário, etc.

A criação básica de um método segue a sintaxe:

```
Sub NomeDoMétodo(ListaDeParametros)
    instruções
End Sub
```

Neste momento não estaremos diferenciando escopo, o que será analisado no módulo 6.

A palavra chave *Sub* indica que o método não retorna dados, ele apenas processa. Caso o método deva retornar um valor, por exemplo, no local de *Sub* se usaria *Function*.

A chamada de um método é feita por colocar o nome mais a lista de parâmetros que ele contenha. Nos exemplos abaixo isto será demonstrado.

#### 5.1.1 Métodos Compartilhados sem Retorno

Métodos compartilhados são aqueles que possuem a execução independente da criação de uma classe por outra, são métodos que não podem ser multi-instanciados.

Métodos sem retorno são utilizados para lidar com variáveis estáticas.

O exemplo abaixo cria um método *Iniciar* e um método *Mostrar* que imprime na tela o valor corrente de uma variável:

```
Shared Contador as Integer
Shared Sub Iniciar()
    Contador += 1
    Mostrar()
End Sub
'Método Mostrar
Shared Sub Mostrar()
    Contador += 1
    Console.WriteLine(Contador.ToString())
End Sub
```

Neste exemplo o método *Iniciar* utiliza uma variável chamada *Contador* que contém um número inteiro compartilhado. Após somar mais um na variável o método *Iniciar* chama o método *Mostrar* que utiliza a mesma variável *Contador*. Isto é possível porque métodos compartilhados só criam e enxergam variáveis do tipo compartilhadas.

Variáveis compartilhadas são aquelas que ao serem criadas só deixam de existir quando a classe onde foram criadas são fechadas. Este tipo de variável não pode ser utilizado fora da classe em que foram criadas.

#### 5.1.2 Métodos Compartilhados com Retorno

Reescrevendo o código anterior agora retornando um valor, o nosso exemplo ficaria:

```
Shared Contador as Integer
Shared Sub Iniciar()
    Contador += 1
    Dim Resultado As Boolean
```

```

        Resultado = Mostrar()
    End Sub
'Método Mostrar
Shared Function Mostrar() As Boolean
    Try
        Contador += 1
        Return True
    Catch
        Return False
    End Try
End Sub

```

Neste exemplo podemos notar que a função *Mostrar* não é mais *sub* e sim *function*, indicando que agora ela tem como saída, ou retorno a quem a chamou, um valor verdadeiro ou falso (*Boolean*), indicando que o método executou ou não com sucesso.

Da mesma maneira, notamos mudanças no método *Mostrar* que dentro do *Try* retorna um *True* indicando que não ocorreu erro ou, caso ocorra erro, retornando *False*. Este valor permite ao método *Iniciar* que fez a chamada saber se o código interno da função *Mostrar* executou corretamente ou não.

Note que no exemplo do método foi utilizada a instrução *Return* que tem a função de fechar o método. Caso após o *Return* ainda existissem linhas de código, estas seriam ignoradas, pois o método teria terminado.

## 5.2 Recebendo Parâmetros

Agora que já abordamos e conseguimos receber um retorno de métodos, precisamos enviar dados ao método. Nos exemplo anteriores o método *Mostrar* utilizava uma variável de nome *Contador* que já estava criada em memória.

### 5.2.1 Parâmetros de Entrada (ByVal)

Atualizando nosso exemplo, imagine o mesmo método agora recebendo um valor para multiplicar o contador:

```

Shared Contador as Integer
Shared Sub Iniciar()
    Contador += 1
    Dim Resultado As Boolean
    Resultado = Mostrar(223)
End Sub
'Método Mostrar
Shared Function Mostrar(ByVal Multiplicador As Integer) As Boolean
    Try
        Contador += 1
        Contador *= Multiplicador
        Return True
    Catch
        Return False
    End Try
End Sub

```

É possível notar que na definição do método consta a lista dos parâmetros que serão recebidos, no caso apenas um. Na chamada do método foi informado o valor 223 que será utilizado para multiplicar o contador original.

Juntando todos estes conceitos, vamos exemplificar com um formulário e duas caixas de texto onde serão digitados dois números e os métodos retornaram as operações matemáticas com os números digitados:

```

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim Variavel1 As Integer = CInt(TextBox1.Text)
    Dim Variavel2 As Integer = CInt(TextBox2.Text)
    MsgBox(Soma(Variavel1, Variavel2).ToString())
    MsgBox(Multiplica(Variavel1, Variavel2).ToString())
    MsgBox(Divide(Variavel1, Variavel2).ToString())
    MsgBox(Subtrai(Variavel1, Variavel2).ToString())
End Sub

```

```

Shared Function Soma(ByVal Numero1 As Integer, ByVal Numero2 As Integer) As Long
    Dim Resultado As Long
    Resultado = Numero1 + Numero2
    Return Resultado
End Function

Shared Function Multiplica(ByVal Numero1 As Integer, ByVal Numero2 As Integer) As Long
    Dim Resultado As Long
    Resultado = Numero1 * Numero2
    Return Resultado
End Function

Shared Function Divide(ByVal Numero1 As Integer, ByVal Numero2 As Integer) As Long
    Dim Resultado As Long
    Resultado = Numero1 / Numero2
    Return Resultado
End Function

Shared Function Subtrai(ByVal Numero1 As Integer, ByVal Numero2 As Integer) As Long
    Dim Resultado As Long
    Resultado = Numero1 - Numero2
    Return Resultado
End Function

```

Cada um dos códigos de métodos acima recebe os dois números como inteiros e faz o cálculo devido, retornando um valor do tipo longo.

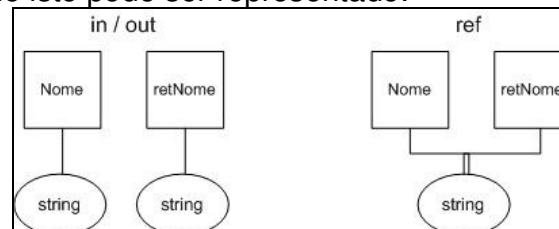
## 5.2.2 Parâmetros Referenciais (ByRef)

Outra forma de retornar valores em parâmetros é utilizar referencial.

O padrão no VB é receber os dados como valor, o que significa que o parâmetro de um método recebe o valor e não o ponteiro. Ao utilizar a instrução *ByRef* no lugar *ByVal* estamos enviando para o método com parâmetros o ponteiro de memória onde a variável original está.

Para simplificar, um parâmetro normal são duas variáveis, uma original e outra recebida pelo método chamado. Quando utilizamos referencial, a variável é a mesma no método original e no método chamado. Quando o método chamado alterar a variável está também alterando no método original.

Veja no diagrama abaixo como isto pode ser representado:



No exemplo com *ByVal* (in) notamos que a variável *Nome* e *retNome* usavam espaços em memória diferente, individualizando os valores. Já no modelo *ByRef* notamos que é um único valor em memória utilizado nos dois métodos. Atualizando o exemplo anterior teríamos:

```

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim Variavel1 As Integer = CInt(TextBox1.Text)
    Dim Variavel2 As Integer = CInt(TextBox2.Text)
    Soma(Variavel1, Variavel2)
    MsgBox(Variavel1.ToString())

```

```

    Multiplica(Variavel1, Variavel2)
    MsgBox(Variavel1.ToString())
    Divide(Variavel1, Variavel2)
    MsgBox(Variavel1.ToString())
    Subtrai(Variavel1, Variavel2)
    MsgBox(Variavel1.ToString())
End Sub

Shared Function Soma(ByRef Numero1 As Integer, ByRef Numero2 As Integer) As Long
    Numero1 += Numero2
End Function
Shared Function Multiplica(ByRef Numero1 As Integer, ByRef Numero2 As Integer) As Long
    Numero1 *= Numero2
End Function
Shared Function Divide(ByRef Numero1 As Integer, ByRef Numero2 As Integer) As Long
    Numero1 /= Numero2
End Function
Shared Function Subtrai(ByRef Numero1 As Integer, ByRef Numero2 As Integer) As Long
    Numero1 -= Numero2
End Function

```

Note que não necessitamos criar uma variável de retorno, pois alteramos e mostramos a primeira variável, que por ter sido referencial foi atualizada.

*DICA: A passagem de parâmetros por valor é mais utilizada, pois a performance é melhor, e não afeta as variáveis originais do método.*

### 5.2.3 Parâmetros Múltiplos

Em algumas situações precisamos passar um número variável de parâmetros.

Nestes casos podemos utilizar array ou enumerador. No caso de array iremos abranger no módulo seguinte.

Quanto ao uso de enumeradores, como já mencionado no módulo 2, se torna útil por permitir que seja utilizado parâmetros nomeados. Quando utilizamos o método normal, se um programador passar parâmetros fora de ordem não temos como detectar. Se for utilizado enumeradores, este risco não existe. Veja o exemplo:

```

Structure eDados
    Dim Código As Integer
    Dim Nome As String
    Dim Telefone As String
End Structure
Shared Sub Iniciar()
    Dim MeusDados As eDados
    MeusDados.Código = 100
    Dim Retorno As Boolean = Dados(MeusDados)
    Console.WriteLine(MeusDados.Nome + " - " + MeusDados.Telefone)
End Sub
Shared Function Dados(ByRef MeusDados As eDados) As Boolean
    If MeusDados.Código = 100
        MeusDados.Nome = "Marcos"
        MeusDados.Telefone = "1234-5678"
        Return True
    Else
        MeusDados.Nome = "Desconhecido"
        MeusDados.Telefone = ""
        Return False
    End If
End Function

```

A grande vantagem no modelo alterado acima é que se o programador inverter a ordem dos dados ao informar os parâmetros não fará diferença, uma vez que cada parâmetro tem seu nome especificado dentro do conjunto de dados.

### 5.3 Overload de Métodos

Após um método ser criado e utilizado, a mudança pode ocasionar problemas, pois as chamadas de um método que contem quatro parâmetros ser alterado para receber cinco parâmetros causaria erro nos códigos onde eram apenas quatro parâmetros.

Para isso criamos *overloads* que consiste em ter mais do que uma função com o mesmo nome, mas com os parâmetros alterados. Para isto temos que conhecer o conceito de assinatura de método. Assinatura de método consiste em nome do método, tipo do método, valor de retorno, número e tipo de parâmetros. Uma alteração em qualquer um destes itens alterou a forma como ele deve ser chamado.

Pense no exemplo utilizado caso ele sofra alterações:

```

Structure eDados
    Dim Código As Integer
    Dim Nome As String
    Dim Telefone As String
End Structure
Shared Sub Iniciar()
    Dim MeusDados As eDados
    MeusDados.Código = 100
    Dim Retorno1 As Boolean = Dados(MeusDados)
    Dim Retorno2 As Boolean = Dados(MeusDados, "07000-000")
    Console.WriteLine(MeusDados.Nome + " - " + MeusDados.Telefone)
End Sub
Shared Function Dados(ByRef MeusDados As eDados) As Boolean
    If MeusDados.Código = 100
        MeusDados.Nome = "Marcos"
        MeusDados.Telefone = "1234-5678"
        Return True
    Else
        MeusDados.Nome = "Desconhecido"
        MeusDados.Telefone = ""
        Return False
    End If
End Function
Shared Function Dados(ByRef MeusDados As eDados, CEP As String) As Boolean
    If MeusDados.Código = 100
        MeusDados.Nome = "Marcos"
        MeusDados.Telefone = "1234-5678"
        Return True
    Else
        MeusDados.Nome = "Desconhecido"
        MeusDados.Telefone = ""
        Return False
    End If
End Function

```

Note que existem dois métodos chamados *Dados*, o primeiro com um parametro e segundo com dois parâmetros. Nas execuções não precisamos definir qual dos métodos será executado, pois o próprio CLR se encarrega de escolher o método que se encaixe no modelo de chamada que foi utilizado.

Com este recurso podemos ter vários métodos com o mesmo nome, mas com uma lista de diferentes parâmetros. Este recurso é extensamente utilizado no próprio framework, visível quando digitamos o nome do método e aparece uma lista das variações que o método possui.

Uma boa prática ao utilizar *overload* é que os métodos chamem uns aos outros, por exemplo, o método que recebe quatro parâmetros e é o mais completo é chamado pelo que recebe apenas dois parâmetros, que chama o de quatro parâmetros passando dois parâmetros vazios. A este processo chamamos de métodos recursivos, ou recursividade.

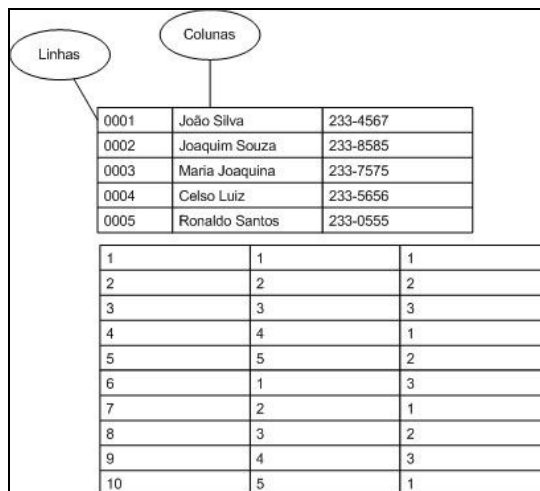
## 6 Arrays e Parâmetros

### 6.1 Definição

Um array não pode ser comparado a um banco de dados como muito o fazem.

Um banco de dados é uma estrutura multi-tipos e bi-dimensional. Arrays são estrutura de um único tipo CTS e multi-dimensional.

Veja o diagrama comparativo:



No primeiro diagrama temos um conjunto de linhas com colunas predefinidas, onde para cada linha temos três colunas específicas e sempre preenchidas, ou seja um plano em 2D.

Já o segundo diagrama mostra um conjunto de 10 linhas consecutivas onde para cada linha temos 2 conjuntos diferentes de números não concorrentes em estrutura 3D. No caso o array seria definido como 10 x 5 x 3.

#### 6.1.1 Definindo Arrays

Para criar um array lembre-se de que só pode conter um tipo. Este deve ser definido logo na definição, como a sintaxe abaixo:

```
Dim NomeDaVariável(posições) As tipo
```

Como exemplo podemos criar um array que contenha a lista de meses:

```
Dim Meses(11) As String
```

```
Meses(0) = "Jan"
```

```
Meses(1) = "Fev"
```

```
Meses(2) = "Mar"
```

```
Meses(3) = "Abr"
```

```
Meses(4) = "Mai"
```

```
Meses(5) = "Jun"
```

```
Meses(6) = "Jul"
```

```
Meses(7) = "Ago"
```

```
Meses(8) = "Set"
```

```
Meses(9) = "Out"
```

```
Meses(10) = "Nov"
```

```
Meses(11) = "Dez"
```

```
Console.WriteLine(Meses(DateTime.Now.Month-1))
```

**DICA:** Ao criar os arrays note que definimos 11 posições, pois os arrays em VB são definidos não em número de ocorrências, mas no valor do índice que deseja.

Para retornar a descrição do mês desejado basta digitar o nome da variável e a posição desejada, como a ultima linha do código de exemplo.

Para criar um array multidimensional a sintaxe é similar:

*Dim NomeDaVariável(PosiçãoX, PosiçãoY, PosiçãoZ) As Tipo*

Para criarmos um array similar a uma tabela de cadastro, veja o exemplo:

*Dim Dados(2,1) As String*

*Dados(0,0) = "Joao"*

*Dados(0,1) = "223-6767"*

*Dados(1,0) = "Joaquim"*

*Dados(1,1) = "223-4545"*

*Dados(2,0) = "Maria"*

*Dados(2,1) = "223-1212"*

Este modelo simula a tabela de cadastro onde utilizamos a primeira posição para indicar a linha e a segunda posição para indicar as colunas. A estrutura criada graficamente representada:

	0	1
0	João	223-6767
1	Joaquim	223-4545
2	Maria	223-1212

## 6.2 Métodos Comuns

Alguns métodos dos arrays são importantes serem considerados, levando-se como exemplo o array *Dados(30,60)*:

Método	Exemplo	Resultado
GetLength	<i>Dados.GetLength(1)</i>	60
Length	<i>Dados.Length</i>	1800
GetValue	<i>Dados.GetValue(5,6)</i>	Nome
GetType	<i>GetType()</i>	System.String[,]
Initialize	<i>Initialize()</i>	Recria todas as posições

Outro interessante método do array permite utilizar o construtor para poder criar o array no momento de definição. O exemplo abaixo demonstra como fazê-lo:

*Dim Números() As Integer = New Integer() {10,20,30}*

*Dim Números(,) As Integer = New Integer(,) {10,20,30}, {1,2,3}*

Estes dois exemplo criam o array já preenchido, o primeiro com as três posições e o segundo exemplo de 3 x 2 posições.

**DICA:** No VB podemos redimensionar arrays com a instrução *Redim...With Preserve*

## 7 Classes Básicas do .NET Framework

### 7.1 Classe AppDomain

Com esta classe temos acesso a dados importantes do sistema atual.

Veja na tabela abaixo um exemplo utilizando uma aplicação:

Método	Resultado
AppDomain.CurrentDomain.BaseDirectory	Diretório da aplicação
AppDomain.CurrentDomain.FriendlyName	Nome do executável
AppDomain.CurrentDomain.SetupInformation.ConfigurationFile	Arquivo de configuração da aplicação

Com estes dados podemos saber o diretório onde está o executável, o nome do sistema sendo executado no momento e onde se encontra a configuração da aplicação.

Com estes dados em mãos podemos gravar arquivos de configuração da aplicação ou ler este que está no mesmo diretório que o executável.

### 7.2 Classe Security

A classe security é importante para termos informações sobre o usuário que está logado na máquina. Essa informação é útil para saber se o nome do usuário, em aplicativos onde podemos ler o usuário do Windows e utiliza-los mais tarde no momento dos acessos da aplicação:

Método	Resultado
System.Security.Principal.WindowsIdentity.GetCurrent().IsAnonymous	False
System.Security.Principal.WindowsIdentity.GetCurrent().IsAuthenticated	True
System.Security.Principal.WindowsIdentity.GetCurrent().Name	msincic
System.Security.Principal.WindowsIdentity.GetCurrent().AuthenticationType	NTLM

Estes métodos para autenticação fornecem dados em aplicações onde utilizaremos o usuário do próprio Windows para controlar o acesso ao sistema.

Também podemos usa-los para gravar logs e acessar banco de dados quando este trabalhar em modo de autenticação pelo sistema operacional e não por modelo proprietário.

### 7.3 Classe IO

A classe IO é importante para acesso a discos, diretórios, arquivos e principalmente leitura e gravação de textos.

O exemplo abaixo demonstra como utilizar a classe de leitura e gravação de arquivos texto:

```
'Cria um novo arquivo texto informando o local. O true indica que é para acrescentar
Dim GravaLog As New System.IO.StreamWriter(@"C:\Log.txt",true)
'Grava no arquivo uma linha com a data e hora atual
GravaLog.WriteLine(DateTime.Now.ToString())
GravaLog.Close()
'Abre o arquivo texto, a arroba serve para indicar string com caracteres especiais
Dim LerLog As New System.IO.StreamReader(@"C:\Log.txt")
'Cria uma string e coloca todo o conteúdo do arquivo, utilizando ReadLine podemos ler uma única linha
Dim Conteudo As String = LerLog.ReadToEnd()
Msgbox(Conteudo)
LerLog.Close()
```

Também podemos utilizar o IO para controlar diretórios e arquivos como os métodos abaixo demonstram:

Método	Resultado
System.IO.File.Delete(caminho)	Deleta um arquivo
System.IO.File.Copy(origem, destino)	Cópia de arquivos

---

System.IO.File.GetLastWriteTime(caminho)	Última alteração no arquivo
System.IO.Directory.CreateDirectory(caminho)	Cria um subdiretório
System.IO.Directory.Delete(caminho)	Deleta diretório
System.IO.Directory.GetDirectories(caminho)	Array dos subdiretórios de um diretório informado
System.IO.Directory.GetFiles()	Array dos arquivos de um diretório informado

Com estas informações podemos detectar alteração em um determinado arquivo de configuração, saber a lista de arquivos e diretórios, deletar e copiar.

## 8 Classes e Objetos

### 8.1 Definição de Classes e Objetos

Uma classe pode ser definida como um negativo para uma foto.

O negativo não pode ser utilizado quando queremos mostrar algo a alguém, nem utilizar o negativo como prova de algo, pois ele não é considerado concreto. Ao utilizarmos o negativo para gerar uma fotografia revelada concretizamos o objeto que chamamos de foto.

Da mesma forma, cada negativo pode gerar infinitas cópias independentes, permitindo que cada cópia da foto seja distribuída a uma diferente pessoa. Já o negativo não pode ser distribuído, nem a foto copiada a menos que a transforme em negativo novamente.

O mesmo pode ser comparado a uma classe. Classes são bases para objetos, são virtuais. Quando queremos utilizar uma classe precisamos instanciá-la, dando-lhe um nome e utilizando esta instancia e não a classe original.

A grande vantagem de utilização das classes é que ao inferir uma mudança na classe, todos os objetos passam automaticamente a refletir as alterações.

Para exemplificar uma classe pense nas variáveis, uma vez que precisamos criar a variável inteira para poder colocar números dentro dela. Não podemos apenas jogar números no tipo inteiro do CTS. Veja o exemplo abaixo:

*Dim Ano As Integer = 2004*      *'Funciona, uma vez que a variável Ano é do tipo inteiro*  
*Integer = 2004*      *'Não funciona, inteiro não pode guardar dados dentro dele*

Quando utilizamos uma classe precisamos atribuir a ela um novo nome, e não utilizamos mais o nome original (*int*) e sim o nome a ela atribuído (*Ano*). A nome atribuído chamamos de objeto.

Utilizando um exemplo real e concreto de meios de veículos podemos entender melhor os conceitos de OOP, uma vez que um veículo do tipo automóvel é igual a outros automóveis da mesma marca, modelo e ano, mudando apenas os valores atribuídos as características. Ou seja, um automóvel é base de qualquer carro que existe, então ao criar um novo carro podemos partir do automóvel com suas características básicas.

O mesmo acontece com pessoas. João, Maria e Joaquim possuem olhos, cabelos, pernas, braços, sabem andar, correr, comer e assim por diante. O que eles tem em comum é que os três vieram da "classe" pessoa que lhes atribui estas propriedades e métodos.

#### 8.1.1 Abstração e Encapsulamento

O primeiro conceito em OOP é abstração e encapsulamento. Significam que objetos são criados para evitar duplicar esforços, sendo este o principal mérito da programação orientada a objetos.

Antes das linguagens OOP em todos os lugares onde precisasse utilizar dados do veículo era necessário recriar as variáveis e reescrever os códigos que controlam o veículo.

Neste aspecto o encapsulamento ajuda porque ao utilizar a classe veículos automaticamente dentro da classe já estão criadas as variáveis que o carro utiliza, bem como as ações que ele realiza. Como os códigos e definições estão prontos, não é necessário recriar variáveis nem reescrever métodos, ou seja, nos tornamos abstratos, uma vez que não precisamos nos preocupar em como fazer determinada tarefa. Encapsulamento está intrínseco a abstração, pois se o código está pronto não precisamos refazê-lo.

Um exemplo prático é quando utilizamos a função *Msgbox()* do framework. Não precisamos definir um formulário, o ícone, os botões nem o código para montar toda a janela que a mensagem

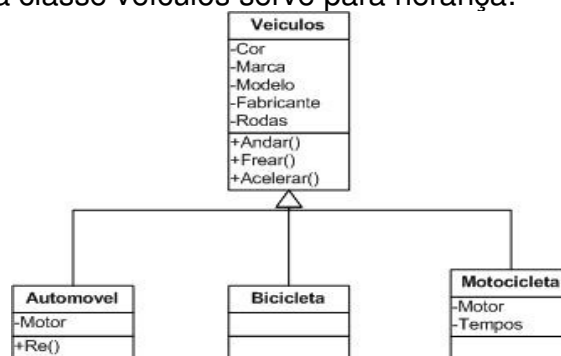
demonstra. Todo o necessário para montar uma mensagem já está encapsulado no *Msgbox*, permitindo que abstraíamos o processo de criar a mensagem.

### 8.1.2 Herança

Conceitualmente herança é quando criamos uma classe de objetos com base em uma já existente, para reaproveitar o que a classe de objetos anterior já possui.

Um exemplo típico de herança é quando criamos uma variável do tipo string. Os métodos que uma string possui não precisam ser criados a todo momento, porque quando criamos uma string herdamos o comportamento do qual todas as strings devem ter.

No gráfico abaixo veja como a classe veículos serve para herança:



Note que todos os veículos possuem características comuns, tanto atributos quanto métodos. Mas automóveis possuem motor e ré que não há nos outros tipos, assim como motocicleta possuiu motor em tempos, o que não acontece com automóvel.

Ao instanciar uma motocicleta não preciso escrever as variáveis cor, marca, modelo, fabricante e rodas, nem os métodos andar, frear e acelerar, já que estes estão sendo herdados do objeto veículos, bastando criar os atributos motor e tempos para ter a motocicleta completa.

### 8.1.3 Polimorfismo

Polimorfismo é permitir que após herdar um determinado método em uma classe eu possa alterá-lo. Como em OOP os códigos estão encapsulados, nem sempre um objeto trabalha exatamente como o objeto original trabalha.

Por exemplo, eu herdei na bicicleta o método acelerar dos veículos, mas uma bicicleta tem um modelo de aceleração diferente dos automóveis e motocicletas. Sendo assim, eu preciso mudar o código encapsulado nos veículos quando o veículo for motocicleta. Ou seja, eu preciso “morfar” o método para adaptá-lo a uma forma diferente da qual originalmente ele deveria ter.

### 8.1.4 Classes Abstratas

Um classe abstrata é quando criamos uma classe que não possa ser diretamente criada. Ela existe para servir de base a outras classes e não para uso direto.

Por exemplo, não existe um veículo, mas sim automóveis, motocicletas e bicicletas. Para que não se utilize veículos a classe é abstrata. Eu posso utilizar a classe veículos como base para a classe automóveis, motocicletas e bicicletas que podem ser utilizados normalmente na aplicação, mas a classe veículos eu não consigo utilizar na aplicação.

### 8.1.5 Interfaces

Estas servem para definir comportamento obrigatório sem definir o código do método.

Todos os veículos precisam saber andar, frear e acelerar, por isso criamos a classe abstrata veículos. Mas além de veículos existe uma superclasse que podemos chamar de “Meios de Locomoção”, e esta inclui barcos, aviões e veículos.


Neste caso não criamos uma classe chamada meios de locomoção pois a variação de atributos é muito grande entre os três tipos mencionados. Neste caso definimos apenas os métodos que todos devem ter, por exemplo, ligar, desligar e acelerar.

Estas são as interfaces, são criadas para obrigar uma classe que a utilize a implementar corretamente o método. Podemos simplificar o conceito de interface por dizer que é um padrão de método a ser utilizado.

Veja abaixo o código que gera uma interface:

```
Interface MeiosDeLocomocao
    Function Ligar() As Boolean
    Function Desligar() As Boolean
    Function Acelerar(Intensidade As Short) As Integer
End Interface
```

A classe que utilizar esta interface será obrigada a criar os três métodos acima e não pode no método acelerar utilizar um longo ao invés de um inteiro curto. Como comentado anteriormente, obrigamos o programador a colocar estas interfaces exatamente como estão definidas.

Representamos interfaces graficamente em UML com o símbolo  MeiosDeLocomocao.

## 8.2 Criação e Instanciamento de Classes

A criação de uma classe segue o padrão definido nos módulos iniciais.

Como exemplo utilizaremos uma classe chamada de pessoas e a partir desta criar os integrantes de um departamento, por exemplo. O código básico da classe é:

```
Public Class Pessoas
    Public Nome As String
    Public Departamento As String
    Public Idade As Integer
    Public Endereco As String

    Public Function Comer(ByVal Alimento As String) As Boolean
        If Alimento.Length = 0 Then
            Return False
        End If
        Console.WriteLine("Estou comendo {0}", Alimento)
        Return True
    End Function

    Public Function Andar(ByVal Intensidade As Integer) As Boolean
        If Intensidade = 0 Then
            Return False
        End If
        Console.WriteLine("Estou andando a {0} passos", Intensidade)
        Return True
    End Function
End Class
```

Analisando o código acima podemos notar que pessoas possuem características como nome, endereço, idade e departamento, assim como os métodos andar e comer. Como os dois métodos já possuem códigos de validação, utilizamos encapsulamento, pois quem utilizar a classe *pessoas* não precisa digitar código de validação ao andar. Também utilizamos abstração por evitar que o

programador tenha que se preocupar em como é feita a validação do andar e comer, isto não importa para ele, pois já está pronto.

Para utilizar uma classe precisa instanciá-la, e fazemos isto por utilizar a seguinte sintaxe básica:

```
Dim NomeDaInstancia As New classe()
```

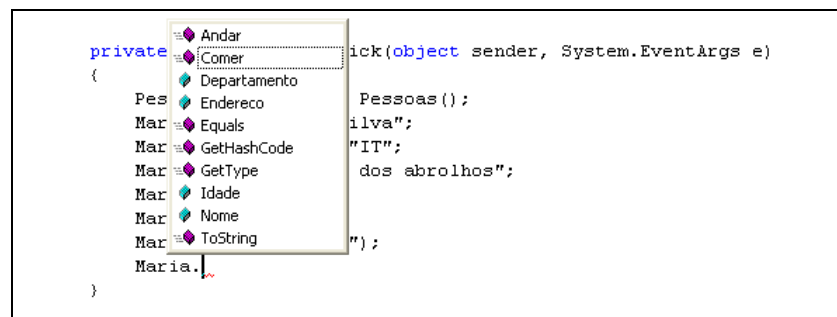
Seguindo esta sintaxe, para utilizar a classe *peessoas* e criar *Maria* utilizamos o código:

```
Dim Maria As New Pessoas()
```

Em seguida, com a instancia *Maria* de *Pessoas* podemos atribuir os valores e utilizar os métodos:

```
Maria.Nome="Maria Silva"
Maria.Departamento="IT"
Maria.Endereco="Rua dos abrolhos"
Maria.Idade=25
Maria.Andar(40)
Maria.Comer("Alface")
```

Note que não precisamos fazer nada para utilizar os métodos e atributos, uma vez que no conceito de objetos *Maria* possui tudo que a classe *Pessoas* possuía originalmente, como mostra a figura a seguir:



## 8.2.1 Propriedades e Enumeradores

Ao criar instancias e classes como o exemplo acima, notamos um inconveniente nos atributos.

Apenas definimos que os atributos são do tipo CTS desejado, mas não estamos validando o tipo de departamento que pode ser utilizado.

A primeira forma de melhorar este processo é utilizando propriedades ao invés de variáveis simples. Para isso precisamos criar uma variável interna na classe que irá guardar o valor informado, e criar a propriedade que mostra ou recebe este valor. Ou seja, todas as propriedades que forem criadas precisam ter uma variável interna para guardar seu valor, uma vez que a propriedade é um método. Vamos transformar *Idade* em propriedade, com o resultado a seguir:

```

Public Class Pessoas
    Public Nome As String
    Public Departamento As String
    Public Endereco As String
    Private pldade As Integer
    Public Property Idade() As Integer
        Get
            Return pldade
        End Get
        Set(ByVal Value As Integer)
            If Value < 5 Or Value > 120 Then
                Throw New Exception("Idade Incorreta...")
            Else
                pldade = Value
            End If
        End Set
    End Property

    Public Function Comer(ByVal Alimento As String) As Boolean

```

```

        If Alimento.Length = 0 Then
            Return False
        End If
        Console.WriteLine("Estou comendo {0}", Alimento)
        Return True
    End Function

    Public Function Andar(ByVal Intensidade As Integer) As Boolean
        If Intensidade = 0 Then
            Return False
        End If
        Console.WriteLine("Estou andando a {0} passos", Intensidade)
        Return True
    End Function
End Class

```

Note que agora *Idade* pode ser utilizado em *Maria* como sempre foi, mas internamente a classe agora guarda a idade na variável *pldade* e a anterior variavel se transformou em dois métodos, sendo o primeiro o *Get*, ou seja, quando perguntarmos a idade de *Maria* com *Maria.Idade* o que irá acontecer é a instrução *Return pldade*. Quando alterarmos a idade de *Maria* com *Maria.Idade=20* o que estamos fazendo é rodando um método *Set* que recebe uma variável *Value*, no caso com valor 20, e atribui este valor a *pldade*.

Como *pldade* é *Private* não aparece na lista de propriedades de *Maria*, existindo apenas internamente.

Outra forma de validar dados é utilizando o *Enum* já visto anteriormente, que gera uma lista de valores possíveis. Vamos implementar *Enum* na lista de departamentos, limitando os departamentos que podem ser utilizados:

```

    Public Enum ListaDepartamentos As Integer
        IT
        RH
        Financeiro
        Administracao
    End Enum

    Public Class Pessoas
        Public Nome As String
        Public Departamento As ListaDepartamentos
        Public Endereco As String
    End Class

```

O atributo *Departamento*, antes do tipo string agora é do tipo *ListaDepartamentos*, portanto, não pode mais ser colocado nele qualquer valor, mas apenas os quatro tipos definidos. Para informar o código de *Maria* agora seria:

```

Dim Maria as New Pessoas()
Maria.Nome="Maria Silva"
Maria.Departamento=ListaDepartamentos.IT

```

Utilizando as propriedades e enumeradores conseguimos criar uma classe mais simpática para uso, com restrições e valores simples de serem encontrados.

## 8.2.2 Construtores

Ainda outro facilitador a utilização de classe são os construtores. Todas as classes possuem um construtor padrão vazio, como a seguir:

```

    Public Class Pessoas
        Sub New()
        End Sub
    End Class

```

O método que leva o mesmo nome da classe sem receber parâmetros é o construtor que é executado quando utilizamos a instrução *Pessoas Maria = new Pessoas()*.

Como a classe `Pessoas` possui quatro atributos obrigatórios, podemos reescrever o construtor para que no momento do `new` já sejam informados. Portanto, o construtor passaria a ser:

```
Public Class Pessoas
    Sub New(ByVal pNome As String, ByVal pDepartamento As ListaDepartamentos, ByVal pEndereco As String, ByVal
    pldade As Integer)
        Nome = pNome
        Departamento = pDepartamento
        Endereco = pEndereco
        Idade = pldade
    End Sub

    Sub New()
    End Sub
```

Com este novo construtor podemos continuar utilizando o `new` sem parâmetros, ou então podemos agora construir o objeto já com os dados:

```
Dim Maria As New Pessoas("Maria", ListaDepartamentos.IT, "Rua dos Abrolhos", 25)
```

Um bom exemplo de construtor é o `Msgbox` que possui 12 diferentes construtores com diferentes parâmetros, utilizando `overload`.

### 8.2.3 Destrutores

Assim como o construtor, também existem os destrutores, métodos que são executados ao se terminar de usar o objeto. Pode ser usado para apagar arquivos temporários, fechar conexão e liberar memória.

Construtores podem ser reescritos, destrutores não. O que podemos é adicionar funções ao destrutor usando o código a seguir:

```
Public Class Pessoas
    Protected Overrides Sub Finalize()
        Console.WriteLine("Fui finalizado")
    End Sub
```

Um importante aviso sobre destrutores é que o componente *Garbage Collector* não destrói um objeto assim que ele para de ser usado, mas sim quando a máquina requer recursos e aciona o GC. Por isso, você não terá como garantir o momento em que seu destrutor irá executar.

## 8.3 Controle de Acessibilidade

Acessibilidade ou visibilidade significa quem pode ver uma determinada variável ou método criado dentro da classe. Os principais assessores no C# são:

Accesor	Escopo visível
Private	Apenas dentro do método ou da classe em que foi criado. Veja como exemplo a variável <code>pldade</code> da classe <code>Pessoas</code> criadas anteriormente.
Public	É acessado dentro da classe e fora da classe para quem a instanciar. Veja como exemplo a variável <code>Nome</code> e o método <code>Andar</code> .
Friend	Público dentro do mesmo <code>assembly</code> compilado, mas privado para outros <code>assemblies</code> .

Os assessores mais utilizados são o *Private* para definir métodos e atributos que serão utilizados localmente e *Public* para os métodos e atributos que precisam ser informados ou executados nos objetos.

### 8.3.1 Métodos Compartilhados

Outro assessor especial é o *Shared*. Este tem uma função diferente dos anteriores pois é combinado com o *Public* ou o *Private*, dependendo da visibilidade desejada.

A grande diferença do *Shared* é que a classe não precisa ser instanciada para que eles sejam utilizados. Um exemplo típico de métodos estáticos é o *Show* do *Msgbox*. Não precisamos instanciar um objeto *Msgbox* para utilizar seu método *Show*.

Podemos exemplificar por criar uma classe *IPVA* para cálculos:

```
Public Class IPVA
    Shared Function CalculaIPVA(ByVal Ano As Integer, ByVal Fator As Integer) As Decimal
        Return Ano * Fator
    End Function
    Shared Function MultaIPVA(ByVal Valor As Decimal) As Decimal
        Return Valor * 0.1
    End Function
End Class
```

Como neste caso tanto o construtor quanto os métodos são compartilhados utilizamos diretamente as operações desejadas, como a seguir:

```
Dim Resultado As Decimal = IPVA.MultaIPVA(3456)
```

**DICA:** A este modelo de classes chamamos de *pattern Singleton*.

## 9 Herança e Polimorfismo

Retornando o exemplo de veículos definidos no módulo anterior iremos ver como o conceito de herança e polimorfismo pode ser bem utilizado.

Para uma classe herdar (usualmente chamada de classe concreta) o que outra classe já implementa a instrução *implements* para classes e *Inherits* para *Interfaces*, e na seqüência o nome da classe que queremos herdar. Veja o exemplo de código a seguir:

```
Interface MeiosDeLocomocao
    Function Ligar() As Boolean
    Function Desligar() As Boolean
    Function Acelerar(ByVal Intensidade As Short) As Integer
End Interface

Public MustInherit Class Veiculos
    implements MeiosDeLocomocao
    Public Cor As String
    Public Fabricante As String
    Public Marca As String
    Public Modelo As String
    Public Rodas As Integer
    Function Andar() As Boolean
        Return True
    End Function
    Function Frear() As Boolean
        Return True
    End Function
    Public Function Acelerar(ByVal Intensidade As Short) As Integer Implements MeiosDeLocomocao.Acelerar
        Return True
    End Function
    Public Function Ligar() As Boolean Implements MeiosDeLocomocao.Ligar
        Return True
    End Function
    Public Function Desligar() As Boolean Implements MeiosDeLocomocao.Desligar
        Return True
    End Function
End Class

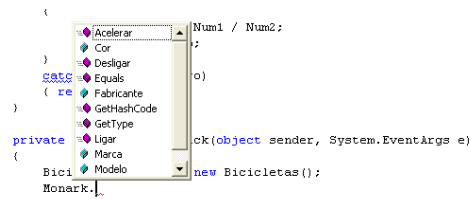
Public NotInheritable Class Bicletas
    Inherits Veiculos
End Class

Public NotInheritable Class Automoveis
    Inherits Veiculos
End Class

Public NotInheritable Class Motocicletas
    Inherits Veiculos
End Class
```

A classe *Veículos* implementa os métodos da interface *MeiosDeLocomocao*, enquanto as classes *Automóveis*, *Motocicletas* e *Bicicletas* herdam a classe *Veículos*, não precisando implementar nada para conter os atributos e métodos da classe pai.

Veja na figura a seguir a utilização simples da classe *Bicicletas*:



Os métodos que a classe *Veículos* possuía, bem como os atributos já constam no momento em que utilizamos a classe *Bicicletas*.

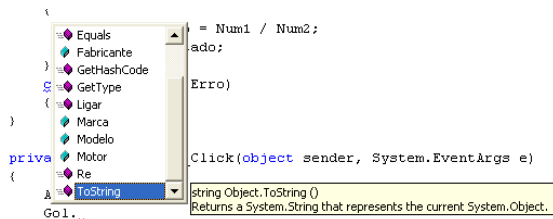
Agora que entendemos a herança precisamos definir o que cada classe tem individualmente, além do que a classe pai herdada já possuía. Como em nosso exemplo as classes *Motocicletas* e *Automoveis* tinham atributos e métodos próprios, seu código seria:

```
Public Class Automoveis
    Inherits Veiculos

    Public Motor As Integer
    Public Function Re() As Boolean
        Return True
    End Function
End Class

Public Class Motocicletas
    Inherits Veiculos

    Public Motor As Integer
    Public Tempos As Integer
End Class
```



Note que além dos métodos e treinamentos já existentes nos veículos, apareceram também o método *Re* e o atributo *Motor* que são próprios de automóveis.

## 9.1 Classes e Métodos Protegidas

Classes protegidas são aquelas que não podem ser herdadas, ou seja, são classes concretas finais. Para isto utilizamos o assessor de proteção temos as instrução *NotInheritable* para a classe e *Protected* para os métodos.

A instrução *NotInheritable* pode ser usada como a seguir:

```
Public NotInheritable Class Automoveis
    Inherits Veiculos

    Public Motor As Integer
    Public Function Re() As Boolean
        Return True
    End Function
End Class

Public NotInheritable Class Motocicletas
    Inherits Veiculos
```

```

    Public Motor As Integer
    Public Tempos As Integer
End Class

```

Com esta alteração não permitimos que um programador crie uma classe a partir da classe automóveis e motocicletas, ou seja se eu tentar executar a instrução abaixo retornará erro:

```

public class Honda
    Inherits Motocicletas

```

Assim como o `NotInheritable` para proteção da herança na classe podemos fazer o mesmo protegendo os métodos de serem alterados por polimorfismo. Para isso utilizamos a instrução `Protected` no método que queremos proteger, como o exemplo abaixo:

```

Public Class Automoveis
    Inherits Veiculos

    Public Motor As Integer
    Protected Public Function Re() As Boolean
        Return True
    End Function
End Class

```

Ao utilizar o `protected` acima não queremos mais a classe protegida, uma vez que neste caso a classe pode ser herdada ou utilizada como classe concreta. A diferença é que o método `Re` só será visto em classes herdadas.

Se no exemplo acima eu criar uma nova classe e herdar a classe `Automoveis` o método `Re` aparecerá, mas se eu criar um objeto utilizando a classe `Automoveis` o método `Re` não irá aparecer, como o exemplo abaixo:

```

public class Automoveis : Veiculos
{
    public int Motor;
    protected bool Re(int Velocidade)
    {
        Console.WriteLine("Estou dando re a {0}", Velocidade);
        return true;
    }
}

sealed public class Motocicletas : Veiculos
{
    public int Motor;
    public short Tempos;
}

sealed public class Veiculos
{
    public int Motor;
    public short Tempos;
}

public class Program
{
    static void Main()
    {
        // ...
    }
}

```

Note que na classe criada que herdou a classe `Automoveis` o método `Re` apareceu, mas apenas na classe que herdou, não sendo visível a objetos criados a partir da nova classe.

## 9.2 Polimorfismo

Utilizamos polimorfismo para alterar o método como algo acontece na classe pai.

Voltando ao exemplo dos veículos, imagine que bicicleta também recebeu um método ligar e desligar, mas precisa mudar a forma destes acontecerem:

```

Public MustInherit Class Veiculos
    implements MeiosDeLocomocao
    Public Cor As String
    Public Fabricante As String
    Public Marca As String
    Public Modelo As String
    Public Rodas As Integer
    Function Andar() As Boolean

```

```
        Return True
    End Function
    Function Frear() As Boolean
        Return True
    End Function
    Public Function Acelerar(ByVal Intensidade As Short) As Integer Implements MeiosDeLocomocao.Acelerar
        Return True
    End Function
    Public Overridable Function Ligar() As Boolean Implements MeiosDeLocomocao.Ligar
        Return True
    End Function
    Public Overridable Function Desligar() As Boolean Implements MeiosDeLocomocao.Desligar
        Return True
    End Function
End Class

Public NotInheritable Class Bicicletas
    Inherits Veiculos
    Public Overrides Function Ligar() As Boolean
        Console.WriteLine("Bicicletas não podem ser ligadas")
        MyBase.Ligar()
    End Function
    Public Overrides Function Desligar() As Boolean
        Console.WriteLine("Bicicletas não podem ser desligadas")
        MyBase.Desligar()
    End Function
End Class
```

Note que os métodos ligar e desligar receberam a palavra chave *Overridable* que identifica aquele método como podendo ser reescrito. Na classe *Bicicletas* os métodos ligar e desligar foram reescritos com a instrução *Override*, identificando que irão sobrepor a implementação original da classe.

A chamada *Mybase.Desligar()* permite que o método desligar reescrito execute o código original da classe pai, após rodar o código alterado.

## 10 Namespace, Delegates e Eventos

### 10.1 Namespace

*Namespaces* podem ser utilizados para organizar métodos e classes.

Por exemplo, o código abaixo cria os métodos com diferentes *Namespaces*:

```
Namespace Politec
    Namespace Utilitários
        Public Class Calculos
        End Class
        Public Class Financeiro
        End Class
    End Namespace
    Namespace Dados
        Public Class CarregaXML
        End Class
        Public Class GravaXML
        End Class
    End Namespace
End Namespace
```

Para utilizar as funções acima podemos utilizar duas formas. A primeira envolve colocar o aplicativo no mesmo *Namespace* que estão as classes ou colocar o *namespace* inteiro na definição do objeto:

```
Namespace Politec.Utilitários
    public class Teste
        Dim x As New Calculos()
        'Classe que esta em outro namespace, diferente da aplicação
        Dim y As New Politec.Utilitários.GravaXML()
    End Class
End Namespace
```

A segunda forma, mais utilizada é definir que irá utilizar o *namespace*, como o exemplo a seguir demonstra:

```
Imports Politec.Utilitários
Imports Politec.Calculos

public Sub Teste()
    Dim x As New Calculos()
    Dim y As New GravaXML()
End Class
```

A vantagem de utilizar o primeiro método de colocar a aplicação no mesmo *Namespace* ou com ele completo é que posso ter métodos com o mesmo nome em *Namespaces* diferentes. Já com a utilização do *Imports* se houver métodos com o mesmo nome, não serão acessados corretamente.

### 10.2 Delegates

Algumas vezes precisamos permitir que um mesmo método chamado pelo programa cliente execute múltiplos métodos diferentes na classe original.

Por exemplo, caso tenhamos vários métodos diferentes que fazem uma determinada função e não queremos em nosso sistema fazer várias condições para chamar o método correto, temos um exemplo de *delegate*.

Pensando em nossa classe de veículos, podemos implementar um *delegate* *Correr* que dispara o *Ligar*, *Desligar* e *Correndo* usando sempre o nome *delegCorrer*. Veja o código atualizado:

```
Public MustInherit Class Veiculos
    implements MeiosDeLocomocao
```

```

Public Delegate Function Correr() As Boolean
Public Cor As String
Public Fabricante As String
Public Marca As String
Public Modelo As String
Public Rodas As Integer
...
Public NotInheritable Class Bicletas
Inherits Veiculos

Dim delegCorry As Correr
Sub New()
    delegCorry = New Correr(AddressOf BicCorrendo)
    delegCorry()
    delegCorry = New Correr(AddressOf Ligar)
    delegCorry()
    delegCorry = New Correr(AddressOf Desligar)
    delegCorry()
End Sub
Public Function BicCorrendo() As Boolean
    Console.WriteLine("Bicletas correndo.")
End Function

```

Este recurso é muito utilizado para no início da aplicação configuramos o *delegate* com o nome do método que irá executar a função desejada, e em qualquer lugar do sistema ao invés de comparar novamente qual é o método, usamos o *delegate* pois este já assumiu a função.

### 10.3 Eventos

Eventos são importantes para definir que algo aconteceu. Como exemplo, quando executamos um formulário o framework nos permite utilizar um evento *Load* que acontece todas as vezes que o formulário foi lido.

O mesmo podemos fazer para que o programador que utilize uma classe saiba que determinada ação na classe aconteceu. Por exemplo, quando o veículo andar automaticamente um evento chamado *EstouAndando* acontece na aplicação cliente.

Para definir um evento, na classe utilize a seguinte sintaxe como o exemplo a seguir:

```

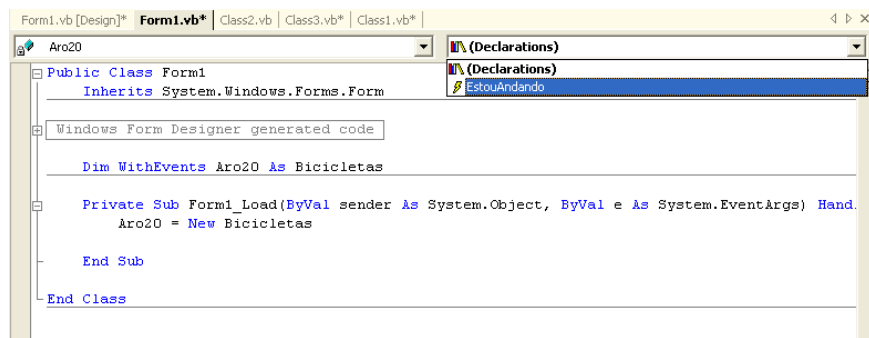
Public MustInherit Class Veiculos
Implements MeiosDeLocomocao
Public Event EstouAndando()

Public Delegate Function Correr() As Boolean
Public Cor As String
Public Fabricante As String
Public Marca As String
Public Modelo As String
Public Rodas As Integer
Function Andar() As Boolean
    RaiseEvent EstouAndando()
    Return True
End Function

```

A primeira linha destacada no código acima mostra como criar um evento, e a segunda linha como explodir a execução do evento no sistema que utilizar esta classe.

Na tela abaixo pode ser visto que agora o objeto derivado de bicicletas possui um evento na lista de eventos do VS:



```
Form1.vb [Design]* | Form1.vb* | Class2.vb | Class3.vb* | Class1.vb*
Aro20
Public Class Form1
    Inherits System.Windows.Forms.Form

    Windows Form Designer generated code

    Dim WithEvents Aro20 As Bicicletas

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Hand
        Aro20 = New Bicicletas

    End Sub

End Class
```

Note que para o evento aparecer na aplicação foi necessário utilizar a cláusula *WithEvents* na criação do objeto.