

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

1	Visão Geral do .NET e Visual Studio 2008	5
1.1	Visão Geral do .NET Framework	5
1.2	Linguagens Suportadas	7
1.3	Resumo das Classes do .NET Framework	7
1.3.1	Class Library	8
1.3.2	Classes Especializadas	9
1.4	Utilizando o Visual Studio 2008	9
2	Visão Geral do C# e Formulários	12
2.1	Conceitos Básicos	12
2.1.1	Regras de Codificação em C#	12
2.1.2	Projetos e Soluções	13
2.1.3	Classes	14
2.2	Formulários e Controles	15
2.2.1	Utilizando Controles	15
2.2.2	Manipulação de Controles	16
2.2.3	Debug	16
2.2.4	Compilação	17
3	Manipulação de Variáveis	19
3.1	Nomeando Variáveis	19
3.2	Tipos de Dados do Sistema	19
3.3	Utilizando Variáveis CTS	20
3.3.1	Criação e Atribuição de Valor	20
3.3.2	Operadores e Precedência	21
3.3.3	Manipulação de String	21
3.4	Conversões Implícitas e Explícitas	21
3.5	Tipos Compostos	22
3.6	Constantes e Read Only	23
3.7	Generics	23
4	Instruções Condicionais, Laços e Desvios	25
4.1	Contexto de Variáveis	25
4.2	Comparativos	25
4.2.1	Comando if	25
4.2.2	Comando switch	26
4.3	Laços	27
4.3.1	Comando while e do	27
4.3.2	Comando for	28
4.3.3	Comando foreach	28
4.4	Desvios	29

5	Tratamento de Erro	30
5.1	Erro de Execução sem Tratamento	30
5.2	Try-Catch-Finally	30
5.2.1	Try-Finally	31
5.2.2	Throw	32
5.3	Checked e Unchecked	32
6	Métodos	34
6.1	Definição de Métodos	34
6.1.1	Métodos Estáticos sem Retorno	34
6.1.2	Métodos Estáticos com Retorno	35
6.2	Recebendo Parâmetros	35
6.2.1	Parâmetros de entrada (in)	35
6.2.2	Parâmetros de saída (out)	37
6.2.3	Parâmetros referenciais (ref)	37
6.2.4	Parâmetros Múltiplos	38
6.3	Overload de Métodos	39
7	Arrays e Parâmetros	41
7.1	Definição	41
7.1.1	Definindo Arrays	41
7.2	Métodos Comuns	42
8	Classes Básicas do .NET Framework	43
8.1	Classe AppDomain e Application	43
8.2	Classe Security	43
8.3	Classe IO	43
9	Classes e Objetos	45
9.1	Definição de Classes e Objetos	45
9.1.1	Abstração e Encapsulamento	45
9.1.2	Herança	46
9.1.3	Polimorfismo	46
9.1.4	Classes Abstratas	46
9.1.5	Interfaces	47
9.2	Criação e Instanciamento de Classes	47
9.2.1	Propriedades e Enumeradores	48
9.2.2	Construtores	49
9.2.3	Destrutores	50
9.3	Controle de Acessibilidade	50
9.3.1	Métodos Estáticos	51
10	Herança e Polimorfismo	52
10.1	Classes e Métodos Protegidas	53

10.2 Polimorfismo	54
11 Namespace, Delegates, Operadores e Eventos	56
11.1 Namespace	56
11.2 Delegates	56
11.3 Operadores	57
11.4 Eventos	58

1 Visão Geral do .NET e Visual Studio 2008

A plataforma .NET não se resume apenas ao framework de desenvolvimento, mas abrange um conjunto de quatro grupos:

- **.NET Framework**

Abrange uma estrutura de objetos, classes e ferramentas que se integram ao sistema operacional para fornecer suporte a desenvolvimento. Ao instalar o .NET Framework em uma máquina não é necessário fazer deployment de outros componentes, uma vez que todos já estão instalados. Isto possibilita fácil desenvolvimento e distribuição de aplicações.

O framework também possibilita que terceiros desenvolvam novos componentes e objetos por utilizar o .NET Framework SDK (Software Development Kit), o que aumenta muito a funcionalidade e os recursos disponíveis.

Podemos comparar o framework também a um sistema operacional para softwares, cabe a ele controlar acesso a memória, instancia e destruição de objetos, variáveis, formulários, gráficos, acesso a disco, etc.

- **.NET Enterprise Servers**

Formada por uma série de servidores preparados com componentes em .NET ou compatíveis, aumenta o poder desta plataforma de desenvolvimento. Veja abaixo a lista dos principais servidores:

Windows Server	Fornecer os recursos de sistema operacional. Fica sobre sua responsabilidade a segurança e autenticação, criptografia, emissão de certificados digitais (PKI), acesso a dados, etc. Um importante componente do Windows Server é o IIS (Internet Information Server) que é o responsável por manter e processar sites web com conteúdo dinâmico ou estático.
SQL Server	Servidor de banco de dados com capacidade até 3 terabytes (3072 gigabytes), gerencia dados com recursos avançados como suporte a XML, XMLHTTP, DTS, etc.
Exchange Server	Gerenciador de correio eletrônico com recursos como distribuição inteligente de documentos, agendamento em grupo, escalabilidade, etc.
Biztalk	Converte, recebe e envia arquivos em XML de qualquer e para qualquer outro sistema.
Commerce Server	Gerenciador de lojas para comércio eletrônico, gerenciando automaticamente promoções, cartão de crédito, entregas, etc.
Host Integrator Server (HIS)	Permite ao SQL Server trocar dados diretamente com mainframes e outros sistemas operacionais como AS/400, OS/390, ISAM, etc.
Internet Security Server (ISA)	Firewall e Proxy para acesso a Internet com lista de usuários, autorização e conteúdo.
Content Manager	Gerenciador de conteúdo para sites de notícias e corporativos, permitindo que usuários finais digitem textos e estes automaticamente sejam formatados na web.
SharePoint	Sistema GED para compartilhamento de arquivos em redes locais ou pela Internet, permitindo mobilidade, controle de versão e atualizações.

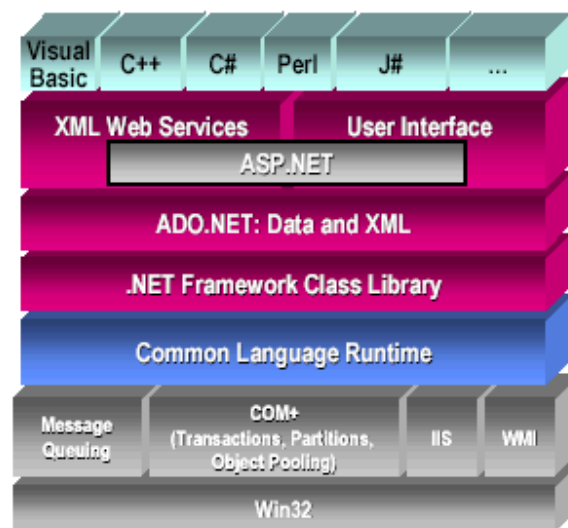
- **Visual Studio .NET**

Ferramenta de desenvolvimento para o framework. Permite rápido desenvolvimento de software por trabalhar com recursos de arrastar e soltar, help on-line, etc. Por ser a ferramenta utilizada neste curso, abrangeremos adiante em detalhes.

1.1 Visão Geral do .NET Framework

Entender o framework é essencial para desenvolvimento neste modelo.

Observe no gráfico abaixo os principais componentes que formam o .NET framework:



- **Linguagens**

Visual Basic.NET, C#, C++.NET, Perl.NET, J# e mais 25 outras linguagens podem ser utilizadas no desenvolvimento. Qualquer linguagem que manipule objetos pode ser utilizada com o framework. Até linguagens antigas como Cobol2, Mumps, Python estão disponíveis em versão para framework.

Em teoria como o framework é um repositório de componentes e objetos, as linguagens citadas funcionam apenas como script, instanciando os objetos que precisam, alimentam suas propriedades e respondem aos eventos.

- **XML Web Services**

Dá suporte a web services que são aplicativos rodando diretamente na Internet.

Antes dos ws precisávamos estar em uma rede local para poder utilizar componentes em uma aplicação, o que era um problema para troca de arquivos e dados entre empresas.

Pensando nisso o consórcio W3C, o mesmo que padronizou a Internet e o html, criou um padrão de comunicações baseadas em xml para troca de dados. Disto surgiram siglas como SOAP, UDDI, WSDL entre outras que dão o suporte à transmissão, publicação e definição respectivamente.

- **User Interface**

Repositório dos controles gráficos utilizados em formulários e páginas web. Dentro deste componente encontramos textbox, label, listbox, grid e todos os outros controles gráficos.

- **ASP.NET**

Gerencia páginas web dinâmicas e também a comunicação dos web services.

- **ADO.NET**

Componente de acesso a dados formados por quatro principais subcomponentes (SQLClient, OracleClient, ODBCClient e OLEDBClient) para suporte a diferentes bancos de dados corporativos.

Sua vantagem é poder escrever os códigos sem a preocupação de diferentes versões do DBMS como acontecia à alguns anos atrás onde as versões de sistemas eram diferentes em cada produto DBMS.

- **.NET Class Library**

Responsável pelos tipos de dados, classes do acesso a disco, thread e todos os serviços da plataforma.

- **Common Language Runtime (CLR)**

Uma definição bem simples e conhecida é máquina virtual. É o CLR que instancia, constrói e

destrói todos os objetos. Podemos dizer que o class library e os outros componentes são é o corpo do framework, mas é o CLR que o faz se movimentar. Abordaremos em seguida o CLR.

Enterprise Services

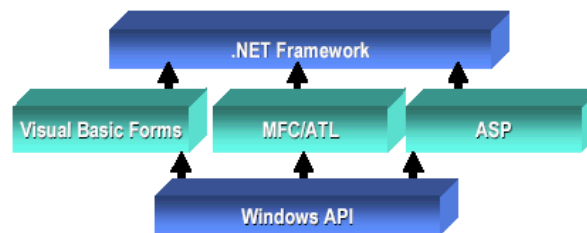
Serviços de suporte a servidores e aplicações, como WMI para acesso a dados do sistema operacional, MSMQ para troca de mensagens, COM+ para controle transacional, etc.

Como referencia, listamos abaixo o conteúdo que será abrangido em cada apostila:

Apostila	Horas	Conteúdo
Implementação de OOP com C#	10	Conceitos de orientação a objetos, introdução ao Framework, sintaxe e OOP utilizando C#.
ASP.NET	16	
ADO .NET	6	Acesso a dados utilizando os modelos conectados (datareader) e desconectado (dataset), XML e XSL Data Document.
.NET Framework Avançado	4	Distribuição e versionamento, acesso a disco, string, remmoting, thread, serialização e integração gerenciado e não gerenciado. Destaca-se a configuração de segurança.
Microsoft Enterprise Services	4	Serviços transacionais com COM+, Message Queue e administração de componentes não gerenciados.

1.2 Linguagens Suportadas

Para visualizar porque o framework é importante e revolucionário no ambiente Microsoft, veja imagem abaixo:



O Visual Basic, o C++ e o ASP eram independentes, o que causava inconsistências, como por exemplo, os componentes do C++ (MFC) eram diferentes dos criados no VB e eram comuns os erros de runtime. Com a unificação de todos os modelos em um único framework, estes problemas foram resolvidos.

No momento da compilação o .NET gera o chamado "pseudocode" que é o código em uma linguagem interpretada que leva o nome de IL (Interpreted Language). Um sistema feito em VB, C#, J# ou qualquer outra ao ser compilado se transforma no mesmo IL.

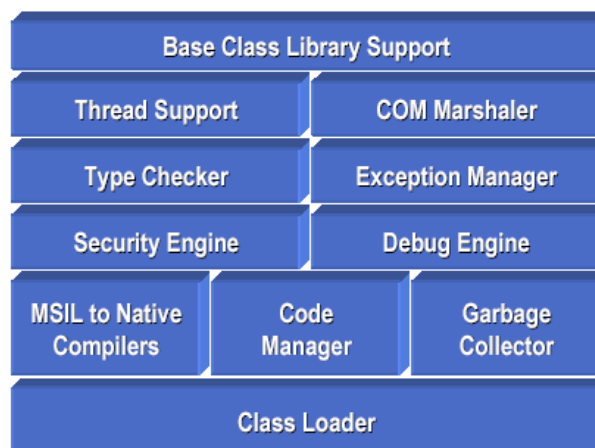
DICA: Você pode Utilize a ferramenta ILDASM.EXE para visualizar o IL.

A única diferença entre as diferentes 30 linguagens é o compilador responsável para transformar os comandos da linguagem em comandos IL, resultando em alguns compiladores sendo mais rápidos que outros. Mas no momento da execução, todos os códigos rodam na mesma velocidade, pois estão na mesma linguagem e utilizando os mesmos componentes, sendo possível um dll em C# executar um programa em Python e assim por diante.

Portanto mais importante do que conhecer a linguagem é conhecer os recursos que o framework do .NET contem para utilizar todos os seus recursos.

1.3 Resumo das Classes do .NET Framework

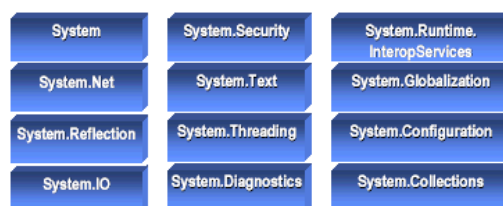
Note diagrama abaixo todos os componentes que formam o CLR e um breve resumo de cada um destes componentes.



Componente	Função
Class Loader	Gerencia os metadados dos objetos, sendo o responsável pela criação e destruição.
MSIL to Native	O .NET trabalha com metadados, ou seja, o código é pré-compila e no momento da execução é gerada em memória uma cópia do componente compilado, que fica ativo enquanto a aplicação estiver ativa.
Code Manager	Gerencia a execução dos códigos, criação de variáveis e tratamento de erros.
Garbage Collector	Responsável por liberação de memória destrói objetos que não estão sendo utilizado. Será tratado em mais detalhes no módulo 6.
Security Engine	Permite ou não a execução e os acessos a disco, servidores, recursos, etc. Abordados em mais detalhes no curso de Framework Avançado.
Debug Engine	Todos os debugs são feitos sobre o código IL, portanto não importa a linguagem utilizada, o debugador é sempre o mesmo.
Type Checker	No .NET variáveis não são criadas na linguagem e sim no framework. Detalhes no módulo 3.
Exception Manager	Contem todas as definições de erro do framework. Detalhes no módulo 5.
Thread Support	Suporte a métodos executados em modo multi-thread fornecendo sincronização. Detalhes no curso Framework Avançado.
Com Marshaller	Componentes criados em .NET podem conversar com COM e vice-versa sobre a orquestração desta classe, fornecendo suporte transacional e outros recursos.
Base Class Library	Integra o código sendo executado com o CLR.

1.3.1 Class Library

Para poder utilizar os componentes do CLR é necessário utilizar o Class Library, já que o CLR fornece a base de execução mas não pode ser diretamente executado. Alistamos abaixo as classes que formam o framework.

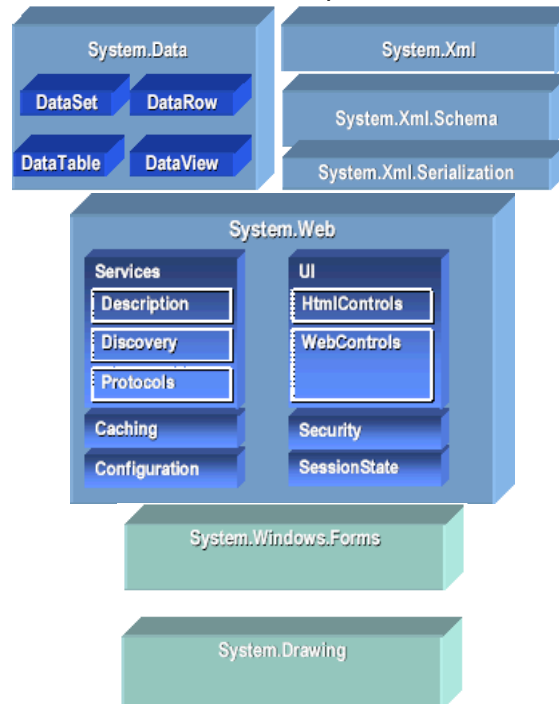


Classe	Função	Abordado em
System	Classe básica de acesso.	
System.Security	Funções de autenticação e autorização por sistema operacional.	Framework Avançado
System.Runtime.InteropServices	Representa o COM Marshaller para execução de e para COM.	Enterprise Services
System.NET	Suporte a TCP/IP, sockets, internet, etc.	Framework Avançado
System.Text	Manipulação de string.	Módulo 3
System.Globalization	Suporte a sistema em multilingüe.	Framework Avançado
System.Reflection	Fazer a leitura dos metadados da classe.	Módulo 11
System.Threading	Suporte a multi-threading.	Framework Avançado
System.Configuration	Leitura de dados do computador e ambiente operacional.	Módulo 8

System.IO	Leitura e escrita de arquivos textos.	Módulo 8
System.Diagnostics	Permite acesso ao Performance Monitor do Windows.	Framework Avançado
System.Collections	Criação de coleções, como arrays.	Módulo 7

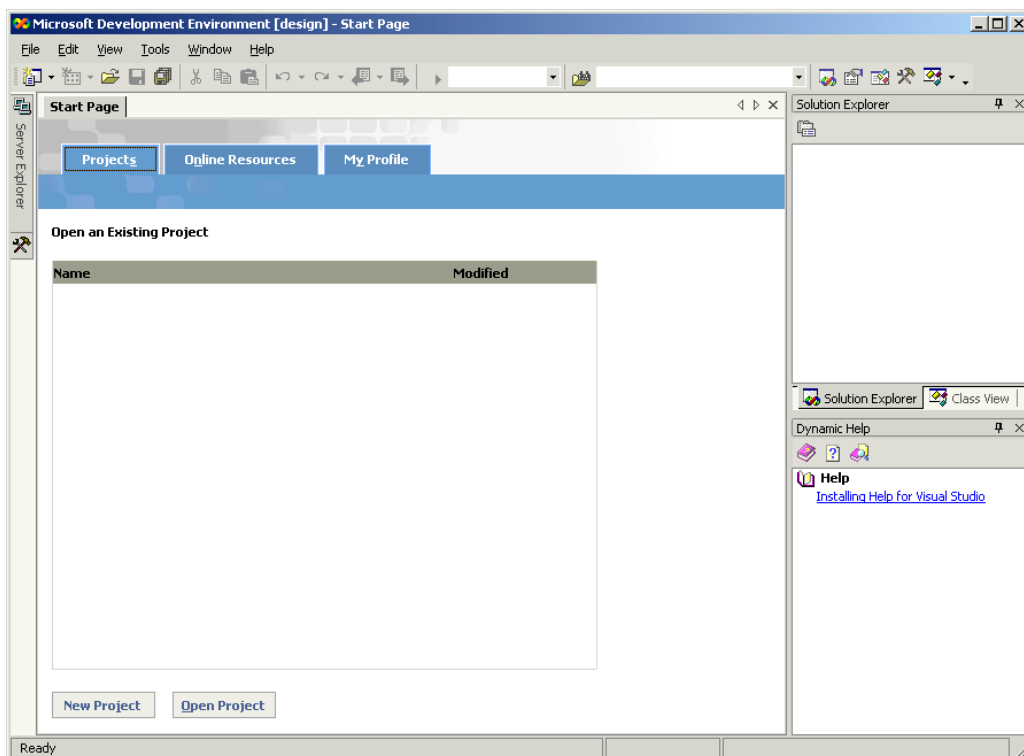
1.3.2 Classes Especializadas

Alem das classes aqui descritas é útil detalhar as classes de dados, web e formulários para melhor aproveitamento. Primeiramente abordamos a classe de dados no gráfico abaixo. A classe de dados será abrangida em detalhes no curso de ADO.NET, as classes de web no curso de ASP.NET e por ultimo as classes de formulários no módulo 2 desta apostila.



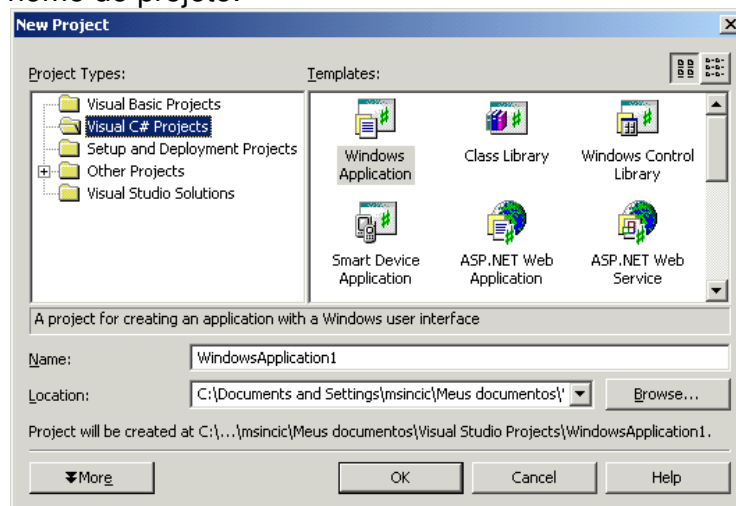
1.4 Utilizando o Visual Studio 2008

Para iniciarmos qualquer projeto é importante conhecer bem o ambiente IDE do VS2008. Abaixo a tela principal do Visual Studio 2008, onde pode-se ver os últimos projetos utilizados ou então clique no botão “New Project”.

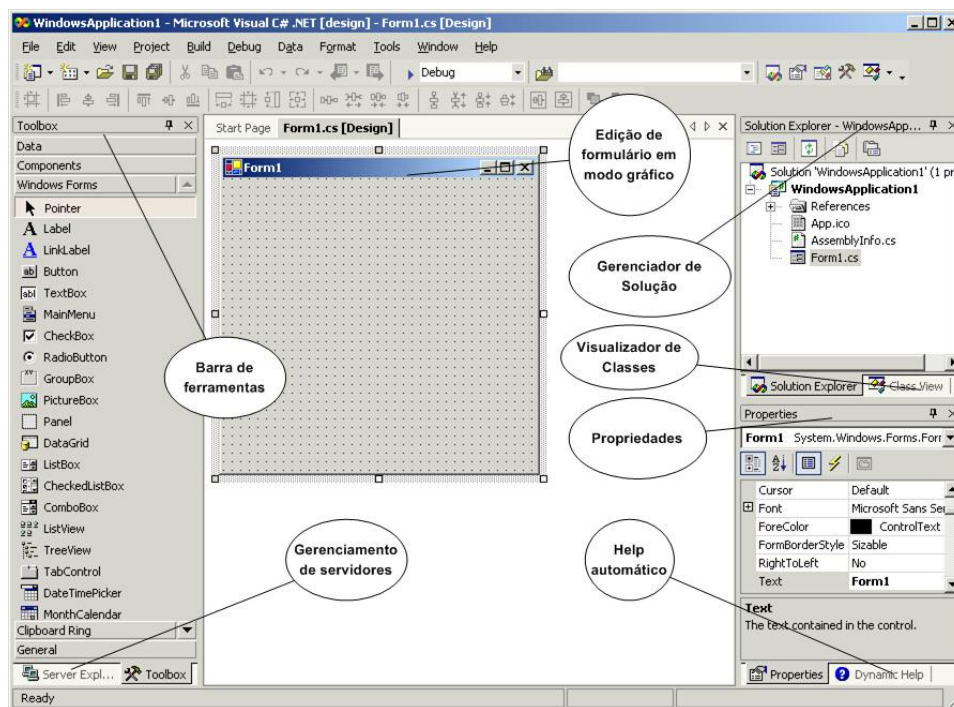


Ao clicar no botão “New Project” a tela abaixo será aberta permitindo escolher a linguagem e o tipo de projeto desejado. Em nosso exemplo iremos utilizar projetos “Windows Application” também chamados de WinForms, projetos desktop.

Coloque no “Name” o nome do projeto e o “Location” padrão é na pasta “Meus Documentos\Visual Studio Projects” mais o nome do projeto.



Ao criar o novo projeto será criado automaticamente o “Form1”. Segue a tela principal do IDE e um breve resumo de como utilizá-las.



Janela	Função
Barra de Ferramentas (Toolbox)	Separada por grupos, utiliza-se por arrastar e soltar no formulário desejado. Alguns tipos de controle como o timer, não aparecer no formulário e sim no rodapé.
Edição de Formulário (Form Editor)	Janela para inclusão e dimensionamento dos controles desejados, bastando arrastar da barra de ferramentas para dentro dele.
Edição de Código (Code Editor)	Janela para edição de código fonte, aberta como tab quando duplo clique sobre qualquer dos objetos do formulário
Gerenciador de Solução (Solution Explorer)	Lista dos formulários, classes e qualquer outro objeto que o projeto possua, aberto para edição com duplo clique.
Gerenciador de Servidores (Server Explorer)	Permite gerenciar Performance Monitor, Message Queue, SQL Server, Services, Event Viewer e conexões criadas no computador local ou remoto se configurado.
Visualizador de Classes (Class View)	Mostra graficamente as classes do projeto com o nome das classes, propriedades e tipos.
Propriedades (Properties)	Alterações dos atributos, mostra as propriedades do objeto clicado no momento.
Help Automática (Dynamic Help)	Mostra automaticamente o help relativo ao comando ou objeto que o ponteiro está locado. É obrigatória a instalação dos CDs do MSDN para funcionar.

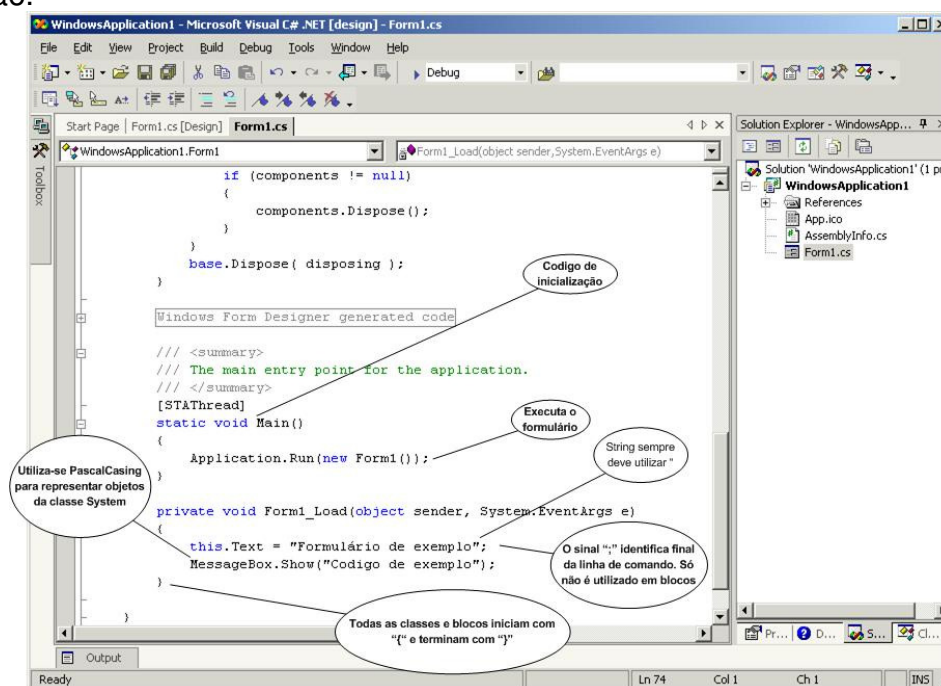
2 Visão Geral do C# e Formulários

2.1 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.

2.1.1 Regras de Codificação em C#

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 de C# podem ser notadas no exemplo:

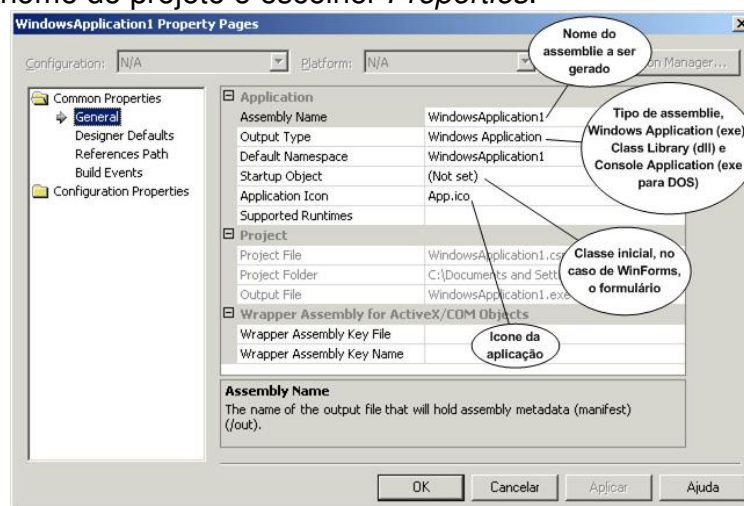
- O método "Main" é obrigatório para a inicialização dos programas em C#, sejam estes formulários, classes, console ou qualquer outro. O C# procura e executa na chamada inicial sempre este método. Por padrão este método faz a função de chamar a classe que irá fornecer a operação, no caso o Form1.
- Todas as linhas de comando precisam ser delimitadas, uma vez que o delimitador de comandos não é o ENTER e sim os sinais ";" ou "{". Veja no exemplo que após todas as linhas existe um destes dois sinais. O sinal ";" é utilizado para indicar que aquele comando termina e que não é em bloco. O sinal "{" indica que aquele comando é um bloco composto.
- 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`. Já o C# como linguagem segue o padrão ECMA, portanto não utiliza nenhum dos dois modelos de

casing por apenas utilizar letras minúsculas. Note que os comandos são todos em minúsculos e apenas as classes utilizam maiúsculos.

- 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 *MessageBox* 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 barras invertidas “\”, que podem ser usadas tanto em uma linha inteira quanto no final da linha. Ao compilar um projeto, os comentários são excluídos do *assembly*.

2.1.2 Projetos e Soluções

No VS todas as aplicações são chamadas de projetos. Cada projeto é um *assembly*, seja este um executável, dll ou aplicação web. O tipo de *assembly* 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 *assembly* 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 assembly e das classes nem no namespace (detalhes no módulo 11).*

O *Startup Object* é um formulário ou classe (formulários no .NET também são classes) que contenha o método *Main*. Notará que nem todos os formulários irão aparecer nesta lista, e quando isto acontecer verifique se o método *Main* está escrito corretamente (veja o exemplo no tópico anterior). 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.

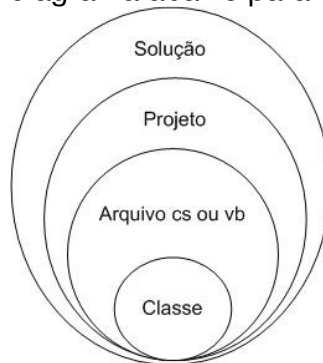
DICA: Solutions não devem ser compartilhados entre usuários, enquanto projetos sim.

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*.

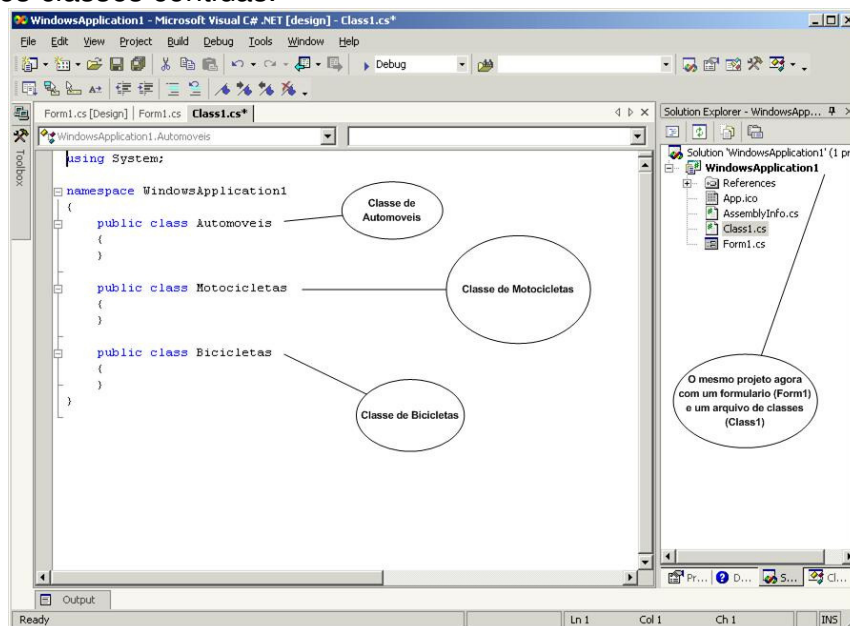
2.1.3 Classes

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 C# 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 *WindowsApplication1*.

DICA: Uma arquivo pode conter múltiplas classes, mas uma classe não pode ser em múltiplos arquivos no framework 1.x.

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. No .NET 2.0 foi introduzido o recurso *partial class* que permite dividir uma mesma classe em mais que um arquivo. Qual a vantagem disto?

Imagine uma classe que monta um formulário com dezenas ou centenas de linhas com definições de variáveis, controles, etc. Podemos dividir a porção de código das definições iniciais e gráficas em um arquivo e os eventos e métodos no segundo arquivo, usando como declaração:

```
public partial class frmClientes
```


























Este recurso pode ser muito bem visto em páginas aspx e formulário onde a porção programador é separada da porções criada pelo IDE. Anteriormente quando este recurso não existia era comum programadores alterarem linhas de código criadas pelo IDE sem intenção e perdermos tempo precioso para conseguir fazer com que o IDE redesenhasse o formulário.

2.2 Formulários e Controles

2.2.1 Utilizando Controles

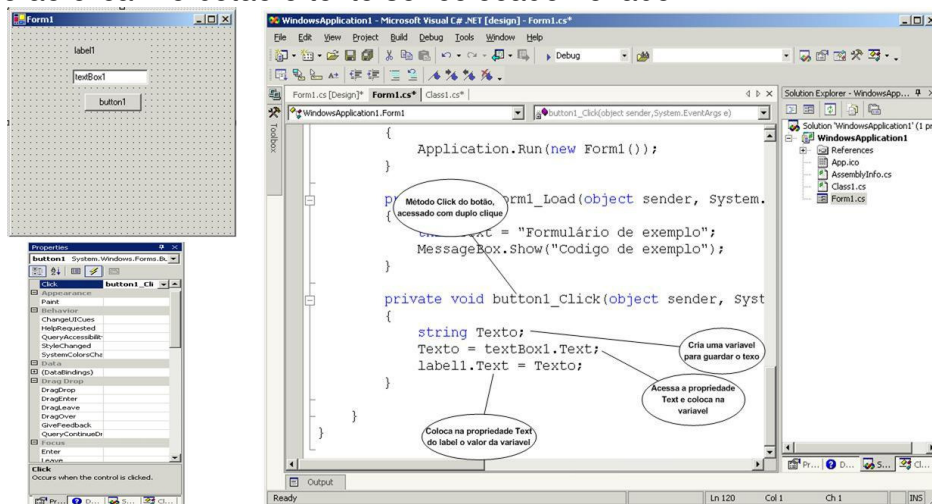
Controles são os objetos gráficos utilizados em formulários.

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 VS2008
 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

2.2.2 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 princípio, 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 podem ser acessados utilizando-se a janela propriedades e clicando-se no botão *Events* (ícone de um raio).

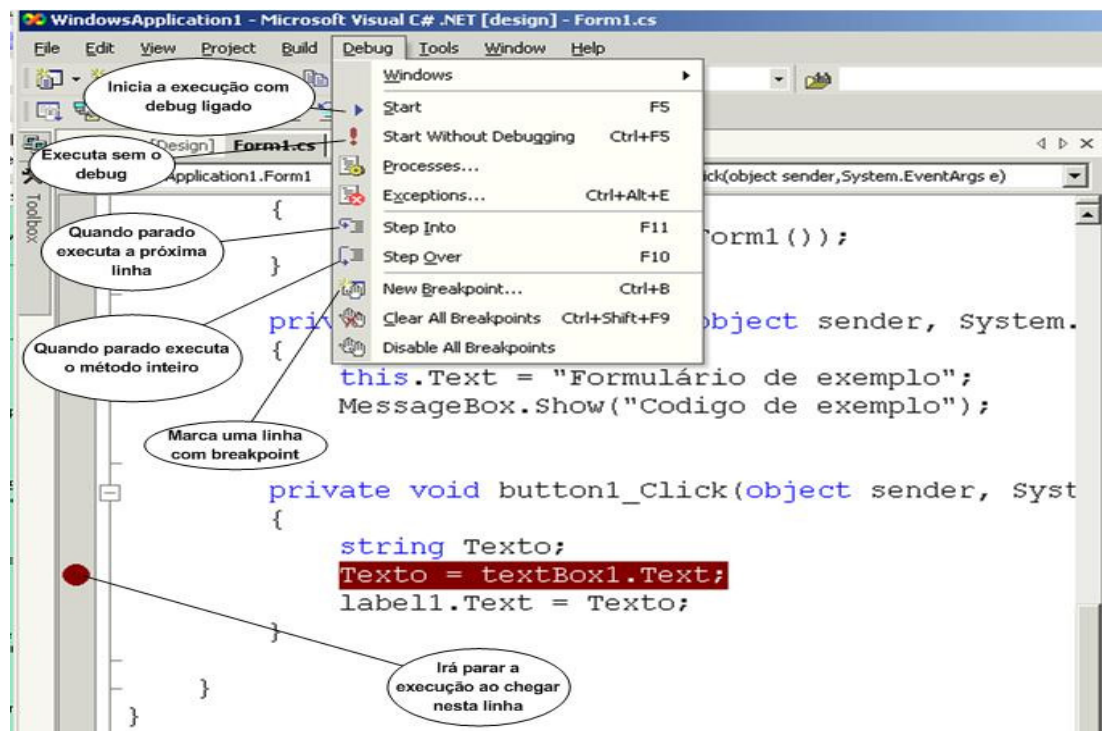
Ao digitar na janela de código o nome do objeto (case sensitivo) automaticamente o VS mostra uma lista com todos os eventos, métodos e propriedades que o controle possui. O símbolo  indica propriedade,  indicam os métodos e  os eventos.

2.2.3 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.

2.2.4 Compilação

Compilar um programa feito em linguagens como Visual Basic 6, C++, 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:

```
csc /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 2008 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.

3 Manipulação de Variáveis

3.1 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. Veja a lista abaixo.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	Goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	Sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
Volatile	while			

3.2 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 C#:

Alias	Nome de Sistema
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal
string	System.String

3.3 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 (in) ou pelo modo referencia (ref). Todas 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*.

3.3.1 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:

```
int Idade; //Variável numérica de nome idade sem valor
Idade = 19; //Atribuído um valor numérico inteiro
string Nome = "Fulano"; //Variável texto Nome com valor já definido
```

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

```
System.Int16 Idade; //Variável numérica de nome idade sem valor
Idade = 19; //Atribuído um valor numérico inteiro
System.String Nome = "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:

```
int Contador = 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, no caso de objetos serve para comparação	Nome = "Fulano"
== e !=	Compara se o valor é igual (==) ou diferente (!=) Para objetos utilize o <i>is</i>	Nome == "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	Idade is int
&&	Significa que as comparações devem ser todas verdadeiras Palavra chave conhecida como "and"	Idade == 30 && Nome == "Fulano"
	Significa apenas uma das comparações precisa ser verdadeira Palavra chave conhecida como "or"	Idade == 30 Nome == "Fulano"
?:	Condicional true/false	Nome=(x-y) ? "OK" : "Erro"
++	Soma um a variável numérica	Idade++
--	Subtrai um à variável numérica	Idade--
%	Divide um numero retornando o resto	Idade % 3 == 0.50
>>= ou <<=	Manipula bits dentro de um byte	1 << (x % 32)

3.3.2 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: *, /, %, +, -, && e ||.

Como alternativa para controle de precedência utilize parênteses. Por exemplo ao invés de escrever $A // B \&\& C$ escreva $(A // B) \&\& C$. 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.

3.3.3 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
IndexOf("a")	Produtividade	10
Insert(3, "xxx")	Maria	Marixxxa
LastIndexOf(s, "a")	Variavel	4
Length	Maria	5
PadLeft(5, "0")	20	00020
PadRight(5, "0")	20	20000
Remove("ar")	Maria	Mia
Replace("ari", "vvv")	Maria	Mvvva
Split(";")	Maria;João;Joaquim	Um em cada posição do array
Substring(2,4)	Produtividade	Odut
ToLower	Produtividade	Produtividade
ToString	Inteiro 60	String 60
ToUpper	Produtividade	PRODUTIVIDADE
Trim	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.

3.4 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:

```
int Numero= 9;
long Convertido = Numero;           //Funciona pois os dois são números
string Convertido = Numero         //Erro, string não é numero
Numero = "888";                     //String não pode ser utilizado em números
```

No primeiro exemplo a variável long é numérica e maior que a variável inteira, portanto não é necessário converter. Já strings podem conter letras, então é necessário validar como numérico para depois converter em numero.

Neste caso utilizamos conversão explicita, como os exemplos:

```
int Numero;
Numero = int.Parse("888")
string Nome;
Nome = Numero.ToString();
```

Note que todos os valores possuem a operação *Parse* para converter valores string ou de outros tipos para o tipo desejado. Outra operação que todos os tipos tem é a operação *ToString* para transformar qualquer valor em string. Veja os exemplos abaixo:

```
int Numero = int.Parse("999")           //Transforma a string em inteiro
long Numero = long.Parse("6767676")    //Transforma a string em long
string Cadeia = 8888.ToString()        //Transforma 8888 em string
```

Para conversão de valores numéricos para outros valores números e de *char* para string podemos utilizar conversão como o exemplo a seguir:


```
long Numero = (long)8777;
int Numero2 = (int)Numero;
```

3.5 Tipos Compostos

Existem dois tipos compostos no .NET Framework. São os valores do tipo *enum* e *struct*.

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
{
    Masculino = 1,
    Feminino = 2,
    Ignorado = 3
}
string NomeFuncionario = "Maria";
int SexoFuncionario = Sexo.Feminino;
MessageBox.Show(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 struct monta uma variável composta por diversos dados. Por exemplo, criar um struct chamado funcionário que contenha nome, idade, telefone e endereço. Veja o exemplo de enum agora combinado com struct:

```
struct Funcionario
{
    string Nome;
    int Idade;
    long Telefone;
    string Endereço;
}
enum Sexo
{
    Masculino = 1;
    Feminino = 2;
    Ignorado = 3;
}
Funcionario Maria;
Maria.Nome = "Maria";
Maria.Idade = 20;
Maria.Telefone = 89897676;
Maria.Sexo = Sexo.Feminino;
MessageBox.Show(Maria.Sexo.ToString());           //Resultado = 2
```

Este exemplo demonstrou muito bem como um struct ajuda, pois ele é utilizado como uma única variável, neste exemplo chamamos de funcionário, e criamos uma variável baseada no struct, no

exemplo Maria, e esta variável contém os atributos que um funcionário deve ter. A vantagem de um struct 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 struct é .

3.6 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 string Aplicacao = "Curso";
const string Cidade = "Taquaritinga";
MessageBox.Show(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 string Aplicacao;
public readonly string Cidade;
Aplicacao = "Curso";
Cidade = "Taquaritinga";
```

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

3.7 Generics

Generics é um recurso interessante para criar variáveis e coleções de dados utilizando formas diferenciadas de ordenação e tipagem de dados. Foram criados para solucionar problemas de coleções que exigiam ordenação em diferentes formatos. Eles residem no namespace

System.Collections.Generic

O primeiro dos 3 principais tipos de *generics* que abordaremos é o *Queue*.

```
System.Collections.Generic.Queue<string> Alunos = new Queue<string>();
Alunos.Enqueue("Marcelo");
Alunos.Enqueue("Joao");
Alunos.Enqueue("Maria");
foreach (string Nome in Alunos)
    Response.Write(Nome + "<br>");
```

Neste primeiro exemplo o resultado será "Marcelo-João-Maria" já que modelos de fila utilizam a ordem de entrada para ordenação no retorno.

O segundo exemplo mostra o *generic* do tipo *Stack*:

```
Stack<string> Alunos2 = new Stack<string>();
Alunos2.Push("Marcelo");
Alunos2.Push("Joao");
Alunos2.Push("Maria");
foreach (string Nome in Alunos2)
    Response.Write(Nome + "<br>");
```

Neste segundo exemplo o resultado será "Maria-João-Marcelo" já que modelos de *stack* são LIFO, utilizam a ordem de entrada inversa para ordenação no retorno.

O terceiro exemplo mostra o *generic* do tipo *SortedDictionary*:

```
SortedDictionary<int, string> Alunos3 = new
SortedDictionary<int, string>();
Alunos3.Add(2, "Marcelo");
```

```
Alunos3.Add(3, "Joao");  
Alunos3.Add(1, "Maria");  
foreach (KeyValuePair<int, string> Nome in Alunos3)  
    Response.Write(Nome.Key.ToString() + "." + Nome.Value + "<br>");  
return;
```

Neste terceiro exemplo o resultado será "1.Maria-2.Marcelo-3.João" já que modelos de stack são ordenados por uma chave primária.

Como vimos nos 3 exemplos (existem vários outros tipos) os *generics* são úteis para tiparmos dados em modo de ordenação rapidamente. Isso pode ser considerado como uma alternativa muito melhor do que arrays, que são de poucos recursos e possuem um range específico de dados quando tipados.

4 Instruções Condicionais, Laços e Desvios

4.1 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, struct 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:

```
int Idade;
{
    int Peso;
    Idade = 30;
    Peso = 60;
}
MessageBox.Show(Idade.ToString());           //Funciona normalmente
MessageBox.Show(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 `MessageBox` ocorreu um erro porque a variável já não existia.

4.2 Comparativos

4.2.1 Comando if

A sintaxe básica do comando `if` é:

```
if(condicao)           //condição verdadeira
    bloco ou comando
else                 //condição falsa
    bloco ou comando
```

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

```
int Idade;
Idade = int.Parse(textBox1.Text);
if(Idade <= 0 || Idade >= 100)
{
    MessageBox.Show("Idade Incorreta. Digite Novamente");
    textBox1.SetFocus();
}
else
    MessageBox.Show("Idade aceita. Obrigado");
```

Note que o `if` contém os colchetes de bloco enquanto o `else` não. O motivo é que quando o `if` ou o `else` (válido para todos os outros comandos deste módulo) tem uma única linha na seqüência não é necessário colocar o bloco. O bloco só é obrigatório quando haverá mais de uma linha após os comandos.

No exemplo utilizamos "`||`" 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:

```
int Idade;
Idade = int.Parse(textBox1.Text);
if(Idade <= 0 || Idade >= 100)
{
    MessageBox.Show("Idade Incorreta. Digite Novamente");
    textBox1.SetFocus();
}
else
{
    string Nome;
    Nome = textBox2.Text;
    if(Nome.Length < 5 || Nome.Length >30)
    {
        MessageBox.Show("Nome Incorreto. Digite Novamente");
        textBox2.SetFocus();
    }
    else
        MessageBox.Show("Dados aceitos.");
}
```

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 switch.

4.2.2 Comando switch

A sintaxe básica do comando switch é:

```
switch(variável)
{
    case 1:
    {
        instrução;
        break;
    }
    case 2:
    case 3:
    case 4:
    {
        instrução;
        break;
    }
    default:
    {
        instrução;
        break;
    }
}
```

Para utilização do *switch* 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 terminando em dois pontos. Assim como o *if*, o *case* só necessita das chaves quando possui mais do que uma linha de comando.

A instrução *default* sinaliza a instrução caso nenhum dos *case* anteriores retorne verdadeiro. No exemplo, se o valor for acima de cinco entrará as instruções do *default* já que os *case* não retornaram verdadeiro.

O *switch* não permite múltiplas condições nem intervalo na mesma linha, portanto quando mais de um valor for utilizado é necessário colocar vários *case*, como no exemplo onde os valores dois a quatro executam a mesma instrução contida no valor quatro. Outra importante consideração do *switch* é a presença do *break* a cada uma das instruções *case* ou *default*. Isto acontece porque diferente do *if* o *case* continua executando quando uma instrução já retornou verdadeiro. O compilador do C# permite a omissão do *break*, mas o VS valida esta instrução não permitindo a compilação.

Veja abaixo um exemplo comparando o tamanho do texto digitado:

```
Texto = textBox1.Text; //Atribui a variável
switch(Texto.Length)//Compara o tamanho da string
{
    case 10:
    case 11:
    case 12: //Os três valores ocorrem como única condição
    {
        MessageBox.Show("Acima de 10 até 12");
        break;
    }
    case 13:
    {
        MessageBox.Show("Valor é 13");
        break;
    }
    default: //Caso o valor esteja diferente de 10 a 13
    {
        MessageBox.Show("Não encontrei...");
        break;
    }
}
```

4.3 Laços

4.3.1 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:

```
while(condição)
{
    instruções
}
```

A sintaxe básica da instrução *do* é:

```
do
{
    instruções
} 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:

```
int Contador = 1; //Variável inicia em 1
while(Contador > 10) //Este código nunca irá rodar
{
    Console.WriteLine("Rodei...while");
    Contador++;
}
//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++;
} while(Contador < 10);
```

4.3.2 Comando for

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 incrementá-la. Já o comando *for* permite que façamos laços seqüências com poucas linhas. Sua sintaxe básica é:

```
for(cria e inicializa a variável, condição, incrementa)
{
    instruções
}
```

Veja o exemplo anterior utilizando a instrução *for*:

```
for(int Contador = 1; Contador < 10; Contador++)
{
    Console.WriteLine("Rodei..{0}", Contador);
}
```

No exemplo criamos a variável contador do tipo inteiro e valor inicial 1. No segundo parâmetro informamos que o laço acontece enquanto contador for menor do que 10, e na terceira instrução somamos um a variável.

4.3.3 Comando foreach

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 número exato de ocorrências. Quanto queremos comparar coleções utilizamos o *foreach*, 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 *foreach* é:

```
foreach(tipo Variavel in Conjunto)
{
    instruções
}
```

Para exemplificar a diferença entre o *for* e o *foreach* imagine uma coleção com 10 nomes criados abaixo:

```
string[] Nomes = new string[10];
for(int Loop = 1; Loop < Nomes.Length; Loop++)
{
    string NomeCorrente;
    NomeCorrente = Nomes[Loop];
    Console.WriteLine(NomeCorrente);
}
foreach(string NomeCorrente in Nomes)
{
    Console.WriteLine(NomeCorrente);
}
```

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

No segundo, utilizando o *foreach*, para cada string que o conjunto de strings *Nomes* contém é atribuído uma variável *NomeCorrente* e esta é impressa sem a necessidade do índice.

Algumas considerações sobre *foreach* 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 NomeCorrente ganhava o valor da seqüência. Portanto, a variável NomeCorrente alterou de valor dez vezes.

4.4 Desvios

Apesar de ser uma linguagem estruturada o C# possui algumas características de linguagens estruturais como pulos de código em caso de códigos condicionais particionados. As instruções que permitem desvios são *goto*, *break* e *continue*.

A sintaxe básica das instruções de desvio são:

```
int Valor = int.Parse(textBox1.Text);
if(Valor == 10)
    goto Escape1;
else
    goto Final;
Escape1:
{
    Console.WriteLine("Entrei no escape");
    goto Final;
}
Final:
{
    Console.WriteLine("Estou saindo");
}
```

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 *switch* 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 *switch* sem criar métodos adicionais no sistema.

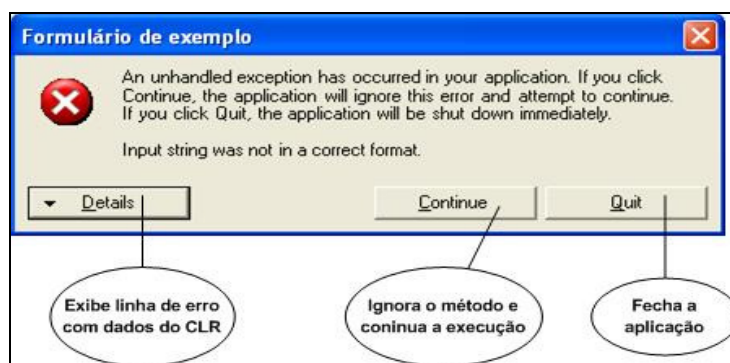
As instruções *break* e *continue* são utilizadas dentro do *switch*, por exemplo, para indicar o final do *case* ou com o *continue* para que a condição seguinte também seja executada.

5 Tratamento de Erro

5.1 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.



5.2 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 (exception Variavel)
{
}
```

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

```
int Numero = 12345;
try
{
    Numero *= 888888;
    Console.WriteLine("Numero alterado.");
}
catch (Exception Erro)
{
    MessageBox.Show("Ocorreu um erro na aplicação. " + Erro.Message)
}
```

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 número. Isto podia ser um problema porque alguns erros eram específicos de banco de dados, acesso, segurança, disco e outros, e para descobrir precisamos comparar o número 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:

```
int Numero = 12345;
try
{
    Numero *= 888888;
    Console.WriteLine("Numero alterado.");
}
catch (OverflowException Estouro)
{
    MessageBox.Show("Valor maior que o possivel.");
}
catch (Exception Erro)
{
    MessageBox.Show("Ocorreu um erro na aplicação. " + Erro.Message)
}
```

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.

5.2.1 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:

```
int Numero = 12345;
try
{
    Numero *= 888888;
    Console.WriteLine("Numero alterado.");
}
catch (OverflowException Estouro)
{
    MessageBox.Show("Valor maior que o possivel.");
}
catch (Exception Erro)
{
    MessageBox.Show("Ocorreu um erro na aplicação. " + Erro.Message)
}
finally
{
    MessageBox.Show(Numero.ToString());
}
```

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

5.2.2 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 Multiplica(int N1, int N2)
    {
        int Resultado;
        try
        {
            Resultado = N1 * N2;
        }
        catch(Exception Erro)
        {
            throw(new Exception("A multiplicação não foi realizada."));
        }
    }
}
```

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

```
public class Teste
{
    static void Main()
    {
        Calculos objMatematico = new Calculos();
        try
        {
            objMatematico.Multiplica(333,444);
        }
        catch (Exception Erro)
        {
            MessageBox.Show(Erro.Message);
        }
    }
}
```

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.

5.3 Checked e Unchecked

Ainda outra forma de tratamento de erro é desconsiderando. Para isto pode-se utilizar o *unchecked* que executa cálculos matemáticos e conversões sem a preocupação com o *cast*, ou seja, se o valor foi ou não convertido, ou *checked* para o efeito contrário. O default do C# é trabalhar em modo *unchecked*, onde não se faz a verificação de overflow aritmético.

A sintaxe das duas instruções é idêntica, mudando-se o nome:

```
checked //ou unchecked
{
    instruções
}
```

}

6 Métodos

6.1 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 clique do botão, o load do formulário, etc.

A criação básica de um método segue a sintaxe:

```
static void NomeDoMétodo(ListaDeParametros)
{
    instruções
}
```

Neste momento não estaremos diferenciando escopo, o que será analisado no módulo 7.

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

A palavra chave *void* indica que o método não retorna dados, ele apenas processa. Caso o método deva retornar um valor, por exemplo, no local de *void* se usaria *int*, *long*, *string* ou qualquer outro valor ou objeto.

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.

6.1.1 Métodos Estáticos sem Retorno

Métodos sem retorno são utilizados para lidar com variáveis compartilhadas.

O exemplo abaixo cria um método *Main* (o primeiro a ser executado) e um método *Mostrar* que imprime na tela o valor corrente de uma variável:

```
static void Main()
{
    int Contador = 0;
    Contador++;
    Mostrar();
}
//Método Mostrar
static void Mostrar()
{
    Contador++;
    Debug.Print(Contador.ToString());
}
```

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

Variáveis estáticas 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.

Também é importante conhecer a instrução *return* que tem como função sair do método a qualquer momento, a ser demonstrado abaixo.

6.1.2 Métodos Estáticos com Retorno

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

```
static void Main()
{
    int Contador = 0;
    Contador++;
    bool ValorRetorno = Mostrar();
    if(ValorRetorno)
        MessageBox.Show("Rodou com sucesso");
    else
        MessageBox.Show("Ocorreram erros...");
}
//Método Mostrar
static bool Mostrar()
{
    try
    {
        Contador++;
        Debug.Print(Contador.ToString());
        return true;
    }
    catch(Exception Erro)
    {
        return false;
    }
}
```

Neste exemplo podemos notar que a função *Mostrar* não é mais *void* e sim *bool*, indicando que agora ela tem como saída, ou retorno a quem a chamou, um valor verdadeiro ou falso, 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 *Main* 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.

6.2 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.

6.2.1 Parâmetros de entrada (in)

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

```
static void Main()
{
    int Contador = 0;
    Contador++;
    bool ValorRetorno = Mostrar(223);
    if(ValorRetorno)
        MessageBox.Show("Rodou com sucesso");
    else
        MessageBox.Show("Ocorreram erros...");
}
```

```
//Método Mostrar
static bool Mostrar(int Multiplicador)
{
    try
    {
        Contador++;
        Contador *= Multiplicador;
        Debug.Print(Contador.ToString());
        return true;
    }
    catch(Exception Erro)
    {
        return false;
    }
}
```

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.

Não é necessário utilizar o parâmetro *in* para indicar que são de entrada, pois este é o tipo padrão dos parâmetros.

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:

```
//Clique no botão
private void button3_Click(object sender, System.EventArgs e)
{
    int Numero1 = int.Parse(textBox1.Text);
    int Numero2 = int.Parse(textBox1.Text);
    //Cria a variavel de Resultado e executa a soma
    long Resultado = Soma(Numero1, Numero2);
    Console.WriteLine(Resultado.ToString());
    Resultado = Subtrai(Numero1, Numero2); //Executa subtração
    Console.WriteLine(Resultado.ToString());
    Resultado = Multiplica(Numero1, Numero2); //Executa Multiplicação
    Console.WriteLine(Resultado.ToString());
    Resultado = Divide(Numero1, Numero2); //Executa Divisão
    Console.WriteLine(Resultado.ToString());
}
static long Soma(int Num1, int Num2)
{
    try
    {
        int Resultado = Num1 + Num2;
        return Resultado;
    }
    catch (Exception Erro)
    { return -1; }
}
static long Multiplica(int Num1, int Num2)
{
    try
    {
        int Resultado = Num1 * Num2;
        return Resultado;
    }
    catch (Exception Erro)
    { return -1; }
}
static long Subtrai(int Num1, int Num2)
{
    try
```

```
        {
            int Resultado = Num1 - Num2;
            return Resultado;
        }
        catch (Exception Erro)
        { return -1; }
    }
    static long Divide(int Num1, int Num2)
    {
        try
        {
            int Resultado = Num1 / Num2;
            return Resultado;
        }
        catch (Exception Erro)
        { return -1; }
    }
}
```

Cada um dos códigos de métodos acima recebe os dois números como inteiros e dentro do tratamento de erro faz o calculo devido. Caso ocorra erro todos os métodos retornam -1, indicativo de que não foi executado.

6.2.2 Parâmetros de saída (out)

Parâmetros de saída podem servir para acessar um banco de dados e retornar mais do que um parâmetro de retorno. Nestes casos precisamos ter uma lista de retorno. Veja o exemplo abaixo:

```
static void Main()
{
    string retNome = "";
    string retTelefone = "";
    bool Retorno = Dados(100, out retNome, out retTelefone);
    Console.WriteLine(retNome + " - "+ retTelefone);
}
static bool Dados(int Codigo, out string Nome, out string Telefone)
{
    if(Codigo==100)
    {
        Nome = "Marcos";
        Telefone = "1234-5678";
        return true;
    }
    else
    {
        Nome = "Desconhecido";
        Telefone = "";
        return false;
    }
}
```

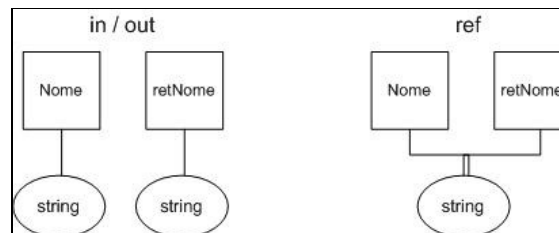
Note que no *Main* criamos duas variáveis vazias e enviamos estas duas variáveis para o método, que recebe as variáveis e preenche seus valores. Neste exemplo ao passar o código o método irá colocar os valores para o nome e o telefone nas variáveis que se chamam *retNome* e *retTelefone* do método original para este imprimir.

6.2.3 Parâmetros referenciais (ref)

Outra forma de retornar valores em parâmetros é utilizar referencial.

O padrão no C# 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 *ref* no lugar *out* 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 utilizando como exemplo o ultimo código demonstrado no *out*:



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

```
static void Main()
{
    string Nome = "";
    string Telefone = "";
    bool Retorno = Dados(100, ref Nome, ref Telefone);
    Console.WriteLine(Nome+" - "+Telefone);
}
static bool Dados(int Codigo, ref string Nome, ref string Telefone)
{
    if(Codigo==100)
    {
        Nome = "Marcos";
        Telefone = "1234-5678";
        return true;
    }
    else
    {
        Nome = "Desconhecido";
        Telefone = "";
        return false;
    }
}
```

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

6.2.4 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 3, 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:

```
struct eDados
{
    int Código;
```

```

        string Nome;
        string Telefone;
    }
    static void Main()
    {
        eDados MeusDados;
        MeusDados.Código = 100;
        bool Retorno = Dados(ref MeusDados);
        Console.WriteLine(MeusDados.Nome + " - " + MeusDados.Telefone);
    }
    static bool Dados(ref eDados MeusDados)
    {
        if(MeusDados.Codigo==100)
        {
            MeusDados.Nome = "Marcos";
            MeusDados.Telefone = "1234-5678";
            return true;
        }
        else
        {
            MeusDados.Nome = "Desconhecido";
            MeusDados.Telefone = "";
            return false;
        }
    }
}

```

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.

6.3 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 no *out* caso ele sofra alterações:

```

    static void Main()
    {
        string retNome = "";
        string retTelefone = "";
        bool Retorno = Dados(100, out retNome, out retTelefone);
        Console.WriteLine(Nome+" - "+Telefone);
        string retCEP = "";
        Retorno = Dados(100, out retNome, out retTelefone, out CEP);
        Console.WriteLine(Nome+" - "+Telefone);
    }
    static bool Dados(int Codigo, out string Nome, out string Telefone)
    {
        if(Codigo==100)
        {
            Nome = "Marcos";
            Telefone = "1234-5678";
            return true;
        }
        else
    }

```

```
        {
            Nome = "Desconhecido";
            Telefone = "";
            return false;
        }
    }
    static bool Dados(int Codigo, out string Nome, out string Telefone, out string CEP)
    {
        if(Codigo==100)
        {
            Nome = "Marcos";
            Telefone = "1234-5678";
            CEP = "08888-999";
            return true;
        }
        else
        {
            Nome = "Desconhecido";
            Telefone = "";
            CEP = "";
            return false;
        }
    }
}
```

Note que existem dois métodos chamados *Dados*, um com três e outro com quatro 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.

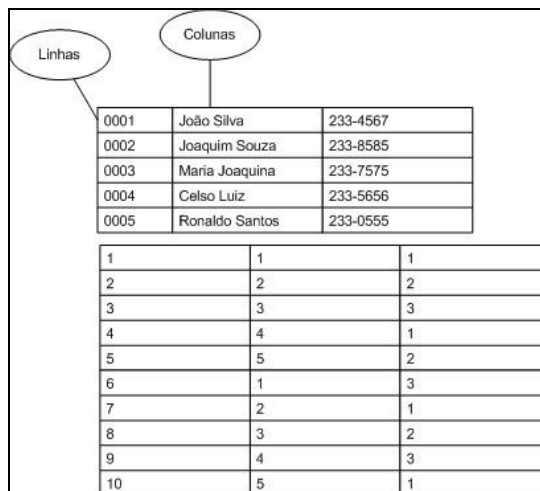
7 Arrays e Parâmetros

7.1 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.

7.1.1 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:

```
Tipo[] NomeDaVariável = new Tipo[Posições]
```

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

```
string[] Meses = new string[12];  
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 12 posições, mas ao indexar utilizamos 11, uma vez que o zero conta como posição.

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:

```
Tipo[,] NomeDaVariavel = new Tipo[PosiçãoX, PosiçãoY, PosiçãoZ, etc.]
```

Note que para cada posição existente no *new* deve haver uma virgula na definição inicial do tipo.

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

```
string[,] Dados = new String[3,2];
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

7.2 Métodos Comuns

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

Método	Exemplo	Resultado
GetLength	Dados.GetLength(1)	60
Length	Dados.Length	1800
GetValue	Dados.GetValue(5,6)	Nome
GetType	GetType()	System.String[,]
Initialize	Initialize()	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:

```
int[] Numeros = new Int[3] {10,20,30}
int[,] Números = new Int[3,2] {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.

8 Classes Básicas do .NET Framework

8.1 Classe AppDomain e Application

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
Application.CommonAppDataPath	Retorna o diretório onde os programas são instalados
Application.CommonAppDataRegistry	Retorna a chave de registry compartilhada
Application.AllowQuit	Define se esta aplicação pode ou não derrubada
Application.CurrentInputLanguage	Configuração regional do usuário
Application.StartupPath	Diretório indicado no atalho
Application.Idle	Evento que acontece quando a aplicação está parada

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.

8.2 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.

8.3 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
System.IO.StreamWriter GravaLog = 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
System.IO.StreamReader LerLog = new System.IO.StreamReader(@"C:\Log.txt");
//Cria uma string e coloca todo o conteúdo do arquivo, utilizando ReadLine podemos ler uma unica linha
string Conteudo = LerLog.ReadToEnd();
MessageBox.Show(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.

9 Classes e Objetos

9.1 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:

```
int Ano = 2004; //Funciona, uma vez que a variável Ano é do tipo inteiro
int = 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.

9.1.1 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 *MessageBox.Show()* do framework. Não precisamos definir um formulário, o ícone, os botões nem o código para montar toda a janela que a

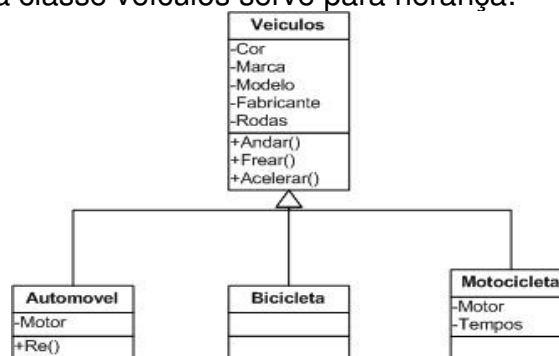
mensagem demonstra. Todo o necessário para montar uma mensagem já está encapsulado no *MessageBox*, permitindo que abstraíamos o processo de criar a mensagem.

9.1.2 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.

9.1.3 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.

9.1.4 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.

9.1.5 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
{
    bool Ligar();
    bool Desligar();
    int Acelerar(Int16 Intensidade);
}
```

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.

9.2 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 string Nome;
    public string Departamento;
    public int Idade;
    public string Endereco;

    public bool Comer(string Alimento)
    {
        if(Alimento.Length==0)
            return false;
        Console.WriteLine("Estou comendo {0}", Alimento);
        return true;
    }

    public bool Andar(int Intensidade)
    {
        if(Intensidade == 0)
            return false;
        Console.WriteLine("Estou andando a {0} passos", Intensidade);
        return true;
    }
}
```

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


```

    public bool Comer(string Alimento)
    {
        if(Alimento.Length==0)
            return false;
        Console.WriteLine("Estou comendo {0}", Alimento);
        return true;
    }

    public bool Andar(int Intensidade)
    {
        if(Intensidade == 0)
            return false;
        Console.WriteLine("Estou andando a {0} passos", Intensidade);
        return true;
    }
}

```

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
{ IT, RH, Administracao, Financeiro }

```

```

public class Pessoas
{
    public string Nome;
    public ListaDepartamentos Departamento;
    public string Endereco;
}

```

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:

```

Pessoas Maria = 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.

9.2.2 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
{
    public Pessoas()
    {
    }
}

```

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
{
    public Pessoas(string pNome, ListaDepartamentos pDepartamento, int pldade, string pEndereco)
    {
        Nome = pNome;
        Departamento = pDepartamento;
        Idade = pldade;
        Endereco = pEndereco;
    }
    public Pessoas()
    {}
}
```

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

```
Pessoas Maria = new Pessoas("Maria", ListaDepartamentos.IT, 25, "Rua dos Abrolhos");
```

Um bom exemplo de construtor é o *MessageBox.Show* que possui 12 diferentes construtores com diferentes parâmetros, utilizando overload.

9.2.3 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
{
    ~Pessoas()
    {
        Console.WriteLine("Fui destruido...");
    }
    public Pessoas(string pNome, ListaDepartamentos pDepartamento, int pldade, string pEndereco)
```

Por estranho que possa parecer, a sintaxe de um destrutor é o nome da classe precedido por um sinal til (~).

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.

9.3 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 pldade da classe Pessoas criadas anteriormente.
public	É acessado dentro da classe e fora da classe para quem a instanciar. Veja como exemplo a variavel Nome e o método Andar.
internal	Público dentro do mesmo assembleie compilado, mas privado para outros assemblies.

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.

9.3.1 Métodos Estáticos

Outro assessor especial é o *static*. Este tem uma função diferente dos anteriores pois é combinado com o *public* ou o *private*, dependendo da visibilidade desejada.

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

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

```
public class IPVA
{
    static IPVA()
    {
    }
    public static decimal CalculaIPVA(int Ano, int Fator)
    {
        return Ano*Fator;
    }
    public static decimal MultaIPVA(decimal Valor)
    {
        return Valor*(decimal)0.10;
    }
}
```

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

```
decimal Resultado = IPVA.MultaIPVA(3456);
```

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

10 Herança e Polimorfismo

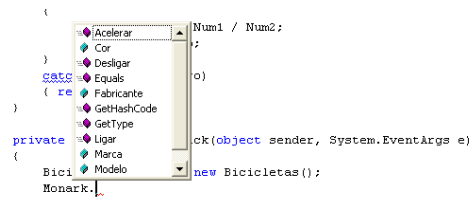
Retornando o exemplo de veículos definidos no módulo anterior iremos ver como o conceito de herança e polimorfismo pode ser bem utilizado.

Para uma classe herdar (usualmente chamada de classe concreta) o que outra classe já implementa utilizamos o sinal dois pontos (:) e na seqüência o nome da classe que queremos herdar. Veja o exemplo de código a seguir:

```
interface MeiosDeLocomocao
{
    bool Ligar();
    bool Desligar();
    int Acelerar(Int16 Intensidade);
}
abstract class Veiculos : MeiosDeLocomocao
{
    public string Cor;
    public string Fabricante;
    public string Marca;
    public string Modelo;
    public int Rodas;
    bool Andar()
    {
        return true;
    }
    bool Frear()
    {
        return true;
    }
    public int Acelerar(Int16 Intensidade)
    {
        return 0;
    }
    public bool Ligar()
    {
        return true;
    }
    public bool Desligar()
    {
        return true;
    }
}
public class Automoveis : Veiculos
{
}
public class Motocicletas : Veiculos
{
}
public class Bicicletas : Veiculos
{
}
```

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*:

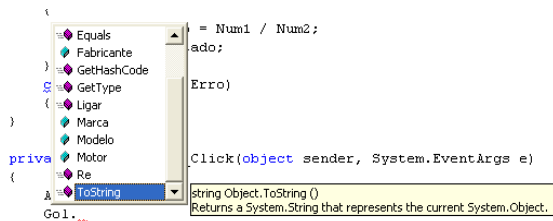


Os métodos que a classe *Veículos* possuía, bem como os atributos já constam no momento em que utilizamos a classe *Bicicletas*.

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

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



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

10.1 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 instruções *sealed* para a classe e *protected* para os métodos.

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

```
sealed public class Automoveis : Veiculos
{
    public int Motor;
    public 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;
}
```

```
}

```

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 : Motocicletas

```

Assim como o *sealed* 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 : Veiculos
{
    public int Motor;
    protected public bool Re(int Velocidade)
    {
        Console.WriteLine("Estou dando re a {0}", Velocidade);
        return true;
    }
}

```

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 *Automóveis* o método *Re* aparecerá, mas se eu criar um objeto utilizando a classe *Automóveis* o método *Re* não irá aparecer, como o exemplo abaixo:

```
public class Automoveis : Veiculos
{
    public int Motor;
    protected public 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 string Marca;
    public string Modelo;
    public bool Re(int Velocidade);
    public string ToString();
}

```

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

10.2 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 abstract class Veiculos : MeiosDeLocomocao
{
    public string Cor;
    public string Fabricante;
    public string Marca;
    public string Modelo;
    int Rodas;
    bool Andar()
    {
        return true;
    }
    bool Frear()
}

```

```
        {
            return true;
        }
        public int Acelerar(Int16 Intensidade)
        {
            return 0;
        }
        public virtual bool Ligar()
        {
            return true;
        }
        public virtual bool Desligar()
        {
            return true;
        }
    }
    sealed public class Bicicletas : Veiculos
    {
        public override bool Ligar()
        {
            Console.WriteLine("Não é possível ligar uma bicicleta.");
            return false;
        }
        public override bool Desligar()
        {
            Console.WriteLine("Não é possível desligar uma bicicleta.");
            return base.Desligar();
        }
    }
}
```

Note que os métodos ligar e desligar receberam a palavra chave *virtual* 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 *base.Desligar()* permite que o método desligar reescrito execute o código original da classe pai, após rodar o código alterado.

11 Namespace, Delegates, Operadores e Eventos

11.1 Namespace

Namespaces podem ser utilizados para organizar métodos e classes.

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

```
namespace Curso
{
    namespace Utilitários
    {
        public class Calculos {}
        public class Financeiro {}
    }
    namespace Dados
    {
        public class CarregaXML {}
        public class GravaXML {}
    }
}
```

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 Curso.Utilitários
{
    public class Teste
    {
        Calculos x = new Calculos();
        //Classe que esta em outro namespace, diferente da aplicação
        Curso.Utilitários.GravaXML y = new Curso.Utilitários.GravaXML();
    }
}
```

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

```
using Curso.Utilitários;
using Curso.Calculos;
{
    public class Teste
    {
        Calculos x = new Calculos();
        GravaXML y = new GravaXML();
    }
}
```

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 *using* se houver métodos com o mesmo nome, não serão acessados corretamente.

11.2 Delegates

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

Por exemplo, se existe um método que disponibiliza determinado dado e faz gravação de log, e todas as vezes em que for chamado o método original este método de log também precisa ser executado, temos um exemplo de *delegate*.

O framework é composto de múltiplos *delegates*, como por exemplo, ao se carregar um formulário ocorrem os eventos *Activate*, *Show*, *Load* e *Initialize*. Basta chamar o delegate *Show* para que os quatro aconteçam juntos.

Pensando em nossa classe de veículos, podemos implementar um delegate *Correr* que dispara o *Ligar*, *Desligar* e *Correndo* de uma única vez. Veja o código atualizado:

```
public abstract class Veiculos : MeiosDeLocomocao
{
    public delegate bool Correr();
    public string Cor;
    public string Fabricante;
    public string Marca;
    ...

    sealed public class Bicicletas : Veiculos
    {
        public Veiculos.Correr DelegCorrendo;
        public Bicicletas()
        {
            DelegCorrendo = new Correr(BicCorrendo);
            DelegCorrendo += new Correr(Ligar);
            DelegCorrendo += new Correr(Desligar);
        }
        public bool BicCorrendo()
        {
            Console.WriteLine("Bicicleta Correndo.");
            return true;
        }
        public override bool Ligar()
        {
            Console.WriteLine("Não é possível ligar uma bicicleta.");
            return true;
        }
        public override bool Desligar()
        {
            Console.WriteLine("Não é possível desligar uma bicicleta.");
            base.Desligar();
            return true;
        }
    }
}
```

Na aplicação de formulário apenas com o código abaixo serão executados os três códigos incluídos no *delegate*:

```
Bicicletas Aro20 = new Bicicletas();
Aro20.DelegCorrendo();
```

Este recurso é muito utilizado quando não queremos disponibilizar o polimorfismo entre objetos, neste caso disponibilizamos o *delegate* para que o programador agrupe o método que ele está criando a um método que a classe pai já possui.

11.3 Operadores

Nos exemplos anteriores notamos que classes podem conter dados, mas podemos levantar uma questão. E se eu precisar somar uma classe a outra?

Para isso podemos fazer *overload* também nos operadores padrão. Por exemplo, se eu criar uma classe de metragem e cada cômodo da casa for uma instancia, precisamos somar as diferentes instancias para conseguir chegar ao resultado desejado.

Veja o exemplo abaixo de como isso pode ser feito:

```
public class Metragem
{
```

```

public decimal Largura;
public decimal Altura;

public static decimal operator+(Metragem Comodo1, Metragem Comodo2)
{
    decimal ResultadoM2;
    decimal M2Comodo1 = Comodo1.Altura * Comodo1.Largura;
    decimal M2Comodo2 = Comodo2.Altura * Comodo2.Largura;
    ResultadoM2 = M2Comodo1 + M2Comodo2;
    return ResultadoM2;
}
}

```

Note que o método de *overload* do operador (+) tem uma sintaxe fixa, onde ele retorna o decimal recebendo as duas instancias desejadas na origem.

Para utilizar o exemplo acima, veja o código:

```

Metragem Sala = new Metragem();
Metragem Quarto = new Metragem();
Sala.Altura = (decimal)2.50; Sala.Largura = (decimal)3.50;
Quarto.Altura = (decimal)3.40; Quarto.Largura = (decimal)2.30;
decimal Apartamento = Sala + Quarto;
MessageBox.Show(Apartamento.ToString());

```

11.4 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 automóvel andar de ré automaticamente um evento chamado *DandoRe* acontece na aplicação cliente.

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

```

public class Automoveis : Veiculos
{
    public delegate void DarRe();
    public event DarRe DandoRe;

    public int Motor;
    public bool Re(int Velocidade)
    {
        Console.WriteLine("Estou dando re a {0}", Velocidade);
        DandoRe();
        return true;
    }
}

```

Note no código que ligado a um *delegate* foi definido que o evento acontecerá, portanto ao chamar a função *Re* do automóvel, a aplicação envia o evento para o aplicativo que chamou o automóvel e o método *ré*.

Como o exemplo especificado é um formulário com um botão, na definição do formulário incluímos os comandos:

```

private void InitializeComponent()
{
    this.Gol = new Automoveis();
    Gol.DandoRe += new Automoveis.DarRe(Gol_DandoRe);
}

```

Os códigos acima criam uma instancia do automóvel e “assinam” o código do evento, indicando que no momento em que o método *DarRe* for executado na classe o evento *Gol_DandoRe* deverá

acontecer no formulário. Portanto, no momento em que algum código mandar o carro andar de ré o evento acontecerá automaticamente.

Abaixo veja o método sendo chamado pelo botão e o evento que responde a ele:

```
private void button5_Click(object sender, System.EventArgs e)
{
    Go.Re(100);
}
private void Go_DandoRe()
{
    MessageBox.Show("O automovel esta dando re...");
}
```