

WTG 2006.2 FSM

Desenvolvendo Aplicações com C# e ASP.NET

Facilitador: José Roberto Araújo

Introdução

.NET é a nova plataforma de desenvolvimento da Microsoft, toda uma nova plataforma de desenvolvimento, o que envolve linguagens de programação, compiladores, modelo de objetos etc., se torna necessária para que consiga englobar de uma forma completamente

Integrada todos esses requisitos. E é essa a proposta de .NET.

A linguagem C# (pronuncia-se C Sharp) faz parte desse conjunto de ferramentas oferecidas na plataforma .NET e surge como uma linguagem *simples, robusta, orientada a objetos, fortemente tipada e altamente escalável* a fim de permitir que uma mesma aplicação possa ser executada em diversos dispositivos de hardware, independentemente destes serem PCs, handhelds ou qualquer outro dispositivo móvel. Além do mais, a linguagem C# também tem como objetivo permitir o desenvolvimento de qualquer tipo de aplicação: Web service, aplicação Windows convencional, aplicações para serem executadas num palmtop ou handheld, aplicações para Internet etc.

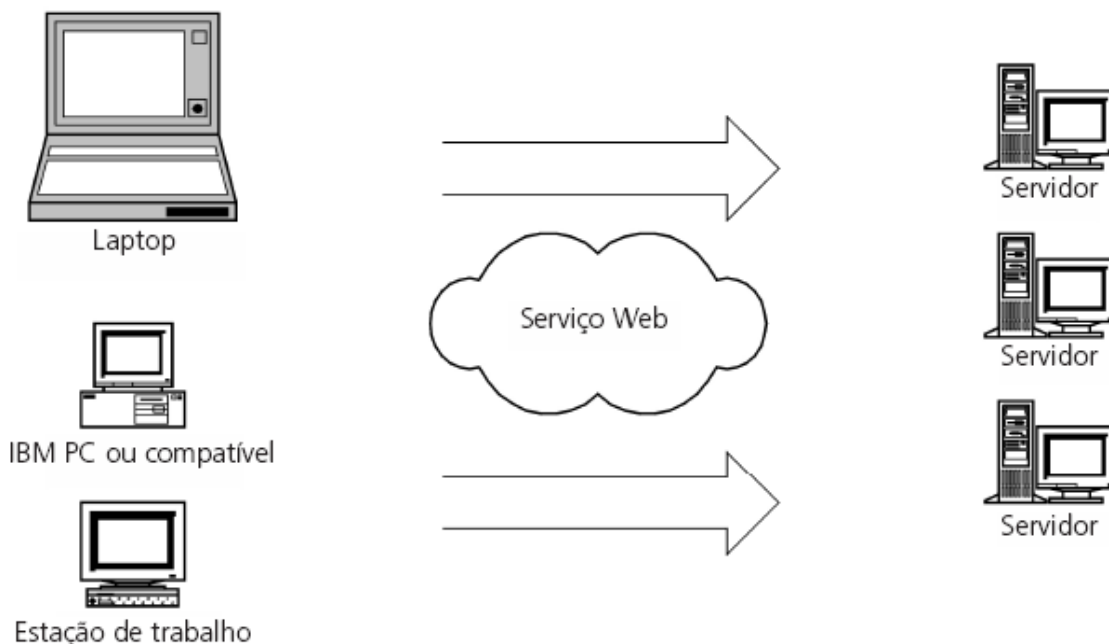


Figura 1.1

Nesta apostila, apresentaremos ao leitor a arquitetura da plataforma .NET a fim de que possa entender onde C# se encaixa nesta plataforma e por que, a despeito da existência de outras linguagens, inclusive aquelas que também dão suporte

a .NET, C# é tão importante.

Atuais dificuldades encontradas no desenvolvimento de sistemas para Windows

Algumas das dificuldades encontradas hoje em dia no desenvolvimento de sistemas são:

- **Complexidade** associada a linguagens de programação de difícil sintaxe, e ainda as dores de cabeça provocadas pelo gerenciamento da memória *heap* por parte do programador.
- **Pouca integração e reaproveitamento** de código entre linguagens de programação diferentes; ausência de implementação de mecanismo de herança entre linguagens diferentes, por exemplo.
- **Diversidade com pouca integração na resolução de problemas complexos**, dificultando a compreensão e o desenvolvimento dos sistemas.
- **Falta de portabilidade de código executável** entre plataformas diferentes.

OBS.: A .NET permite que usemos a linguagem de programação da qual mais temos domínio e mesmo assim continuamos a usufruir todo o seu potencial.

A abordagem .NET

Citaremos a seguir algumas das características de .NET que visam a resolver os problemas citados acima:

- **Independência de linguagem de programação:** o que permite a implementação do mecanismo de herança, controle de exceções e depuração entre linguagens de programação diferentes.
- **Reutilização de código legado:** o que implica em reaproveitamento de código escrito usando outras tecnologias como COM, COM+, ATL, DLLs e outras bibliotecas existentes.
- **Tempo de execução compartilhado:** o runtime do .NET é compartilhado entre as diversas linguagens que a suportam, o que quer dizer que não existe um runtime diferente para cada linguagem que implementa .NET.
- **Sistemas auto-explicativos e controle de versões:** cada peça de código .NET contém em si mesma a informação necessária e suficiente de forma que o runtime não precise procurar no registro do Windows mais informações sobre o programa que está sendo executado. O runtime encontra essas informações no próprio sistema em questão e sabe qual a versão a ser executada, sem acusar aqueles velhos conflitos de incompatibilidade ao registrar DLLs no Windows.
- **Simplicidade** na resolução de problemas complexos.

A Arquitetura .NET

Para melhor entendermos tudo o que temos dito até aqui, vamos falar um pouco da arquitetura de .NET e os seus principais componentes.

CLR (Common Language Runtime)

O CLR, ou tempo de execução compartilhado, é o ambiente de execução das aplicações .NET. Como o leitor já deve ter atentado, as aplicações .NET não são aplicações Win32 propriamente ditas (apesar de executarem no ambiente Windows), razão pela qual o runtime Win32 não sabe como executá-las. O Win32,

ao identificar uma aplicação .NET, dispara o runtime .NET que, a partir desse momento, assume o controle da aplicação no sentido mais amplo da palavra, porque, dentre outras coisas, é ele quem vai cuidar do gerenciamento da memória via um mecanismo de gerenciamento de memória chamado *Garbage Collector* (GC) ou coletor de lixo, acerca do qual falaremos mais tarde. Esse gerenciamento da memória torna os programas menos susceptíveis a erros. Mais ainda, o CLR como seu próprio nome o diz, é compartilhado e, portanto, não temos um runtime para VB.NET, outro para C# etc. É o mesmo para todo mundo.

CTS (Common Type System)

O CTS, ou *Sistema Comum de Tipos*, que também faz parte do CLR, define os tipos suportados por .NET e as suas características. Cada linguagem que suporta .NET tem de, necessariamente, suportar esses tipos. Apesar de que a especificação não demanda que todos os tipos definidos no CTS sejam suportados pela linguagem, esses tipos podem ser um subconjunto do CTS, ou ainda um superconjunto. Um conjunto de classes básicas que define todos os tipos é implementado na CTS. Por exemplo: um tipo *Enum* deve derivar da classe *System.Enum* e todas as linguagens devem implementar o tipo *Enum* dessa forma. Todo tipo deriva da classe *Object*, porque em .NET tudo é um objeto e, portanto, todos os tipos devem ter como raiz essa classe. É dessa forma que os diversos tipos nas diversas linguagens são implementados, obedecendo às regras definidas no CTS. Na .NET, e em C# conseqüentemente, todos os tipos derivam de uma raiz comum: a classe *Object*, o que equivale a dizer que todos os tipos são objetos, por definição.

CLS (Common Language Specification)

O CLS, ou *Especificação Comum da Linguagem*, é um subconjunto do CTS, e define um conjunto de regras que qualquer linguagem que implemente a .NET deve seguir a fim de que o código gerado resultante da compilação de qualquer peça de software escrita na referida linguagem seja perfeitamente entendido pelo runtime .NET. Seguir essas regras é um imperativo porque, caso contrário, um dos grandes ganhos do .NET, que é a independência da linguagem de programação e a sua interoperabilidade, fica comprometido. A grosso modo, dizer que uma linguagem é compatível com o CLS significa dizer que mesmo quando esta é sintaticamente diferente de qualquer outra que implemente .NET, semanticamente ela é igual, porque na hora da compilação será gerado um código intermediário (e não código assembly dependente da arquitetura do processador) equivalente para duas peças de código iguais, porém escritas em linguagens diferentes. É importante entender esse conceito para não pensar que o código desenvolvido em C# não pode interagir com código desenvolvido em VB ou outras linguagens, porque mesmo estas sendo diferentes, todas são compatíveis com o CLS.

BCL (Base Classe Library)

Como era de se esperar, uma plataforma que promete facilitar o desenvolvimento de sistemas precisa ter uma biblioteca de classes básica que alavanque a simplicidade e a rapidez no desenvolvimento de sistemas. É este o objetivo da BCL (*Biblioteca de Classes Base*), oferecer ao desenvolvedor uma biblioteca consistente de componentes de software reutilizáveis que não apenas facilitem, mas também que acelerem o desenvolvimento de sistemas. Na BCL encontramos classes que contemplam desde um novo sistema de janelas a bibliotecas de entrada/saída, gráficos, sockets, gerenciamento da memória etc.

Namespaces	
Namespace	Descrição
System	Contém algumas classes de baixo nível usadas para trabalhar com tipos primitivos, operações matemáticas, gerenciamento de memória etc.
System.Collections	Pensando em implementar suas próprias pilhas, filhas, listas encadeadas? Elas já foram implementadas e se encontram aqui.
System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient	Aqui você vai encontrar tudo o que é necessário para lidar com bases de dados e, como é de se esperar, você encontrará ADO.NET aqui.
System.Diagnostics	Log de Event, medição de performance, classes para gerenciamento de processos, depuração e mais você poderá encontrar neste namespace.
System.Drawing	A .NET oferece uma biblioteca de componentes para trabalhar com gráficos, chamada GDI+, que se encontra neste namespace.
System.IO	Biblioteca para lidar com entrada e saída, gerenciamento de arquivos etc.
System.NET	Aqui você encontra bibliotecas para programação de redes, sockets etc.
System.Reflection	Em .NET você pode gerar código em tempo de execução, descobrir tipos de variáveis etc. As bibliotecas necessárias para isso encontram-se neste namespace.
System.Runtime.InteropServices System.Runtime.Remoting	Fornecem bibliotecas para interagir com código não-gerenciado.
System.Security	Criptografia, permissões e todo o suporte ao qual .NET oferece a segurança você encontra aqui.
System.Threading	Bibliotecas necessárias para o desenvolvimento de aplicações multithread.
System.Web	ASP.NET, Web services e tudo o que tenha a ver com Web pode ser encontrado aqui.
System.Windows.Forms	Bibliotecas para o desenvolvimento de aplicações Windows tradicionais.
System.XML	Bibliotecas que permitem a interação com documentos XML.

Compilando programas .NET: introduzindo a linguagem intermediária MSIL (Microsoft Intermediate Language)

A MSIL - ou simplesmente IL - é a linguagem intermediária para qual é interpretado qualquer programa .NET, independente da linguagem em que este for escrito. Essa tradução é feita para *código intermediário* (como em JAVA com os *byte codes*) sintaticamente expresso na IL. Por sua vez, qualquer linguagem .NET compatível, na hora da compilação, gerará código IL e não código assembly específico da arquitetura do processador onde a compilação do programa é efetuada, conforme aconteceria em C++ ou Delphi, por exemplo. E por que isso? Isso acontece para garantir duas coisas: a independência da linguagem e a independência da plataforma (arquitetura do processador).

Arquitetura .NET

Linguagens de Programação	
Base Class Library (BCL)	
Common Language Runtime (CLR)	
Common Type System	Common Language Specification

OBS.: A MSIL é a linguagem intermediária para qual é interpretado qualquer programa .NET na hora da compilação, independente da linguagem em que este for escrito.

Como uma aplicação .NET é executada pelo Runtime

Para podermos falar sobre este assunto vamos introduzir alguns conceitos essenciais para a compreensão da execução de um aplicativo .NET.

Tempo de Compilação

Entende-se por tempo de compilação a parte do processo de compilação que diz respeito à geração de código em MSIL (linguagem intermediária) e de informações específicas da aplicação necessárias para a sua correta execução. Mas onde estas informações são armazenadas? Como resposta a esta pergunta vamos introduzir o conceito de METADATA ou metadados.

METADADOS

São um conjunto de instruções geradas no processo de compilação de qualquer programa .NET, junto com a MSIL, que contém as seguintes informações específicas da aplicação:

- A descrição dos tipos (classes, estruturas, tipos enumerados etc.) usados na aplicação, podendo esta ter sido gerada em forma de DLL ou de executável
- A descrição dos membros de cada tipo (propriedades, métodos, eventos etc.)
- A descrição de cada unidade de código externo (assembly) usada na aplicação e que é requerida para que esta execute adequadamente
- Resolução da chamada de métodos
- Resolução de versões diferentes de uma aplicação

Dada a informação contida nos METADADOS, podemos dizer que uma aplicação .NET é auto-explicativa, dispensando a utilização do registro do Windows para armazenar informações adicionais a seu respeito. Mais ainda, nos METADADOS é armazenada a versão da aplicação, o que permite que duas aplicações, mesmo sendo homônimas, possam conviver amigavelmente sem gerar conflitos de versão no sistema hospedeiro. Falaremos mais a esse respeito quando abordarmos a discussão de assemblies e namespaces.

O CLR vai procurar nos METADADOS a versão correta da aplicação a ser executada. Esse é um ganho muito grande no que diz respeito à implementação e manutenção de sistemas em produção, dadas as dificuldades associadas à manutenção de DLLs e de componentes cujas versões são diferentes, mas cuja convivência no mesmo ambiente é necessária por razões de compatibilidade com outros aplicativos que precisam de uma ou de outra DLL.

ASSEMBLY

Toda aplicação .NET, quando compilada, é armazenada fisicamente numa unidade de código denominada *assembly*. Uma aplicação pode ser composta de um ou mais *assemblies*, os quais são representados no sistema de arquivos do sistema operacional host na forma de arquivos executáveis, de extensão .EXE, ou de uma biblioteca de ligação dinâmica melhor conhecida como DLL, e obviamente de extensão .DLL.

PE (Portable Executable)

Quando um aplicativo é compilado, são geradas instruções em IL. Como já dissemos acima, METADADOS com informações da aplicação também são gerados, e obviamente armazenados na forma de uma DLL ou de um arquivo executável. Isso é conhecido como *Executável Portável (Portable Executable)* ou simplesmente PE. Diz-se portável porque ele poderá ser executado em qualquer plataforma que suporte .NET, sem necessidade de recompilação, operação que será efetuada automaticamente pelo runtime quando da execução da aplicação.

Compilação JIT ("Just In Time")

Um compilador JIT, também conhecido como JITTER, converte instruções IL para instruções específicas da arquitetura do processador onde a aplicação .NET está sendo executada. Na plataforma .NET existem três diferentes tipos de JITTER:

- *Pré-JIT*: Compila de uma só vez todo o código da aplicação .NET que está sendo executada e o armazena no cache para uso posterior.
- *Econo-JIT*: Este tipo de compilador é usado em dispositivos como handhelds onde a memória é um recurso precioso. Sendo assim, o código é compilado sob demanda, e a memória alocada que não está em uso é liberada quando o dispositivo assim o requer.
- *Normal-JIT*: O Normal-JIT compila o código sob demanda e coloca o código resultante no cache, de forma que esse código não precise ser recompilado quando houver uma nova invocação do mesmo método.

VES (Virtual Execution System)

O processo de compilação acontece num ambiente chamado de *Sistema de Execução Virtual (VES)*, e é aqui onde o JITTER é ativado quando uma aplicação .NET é chamada. O JITTER é ativado a partir do runtime do Win32, passando o controle para o runtime .NET; após isso, a compilação do PE é efetuada e só então o código assembly próprio da arquitetura do processador é gerado para que a aplicação possa ser executada.

O diagrama a seguir ilustra todo o processo de execução de uma aplicação, desde a geração das instruções IL em tempo de compilação, até a geração do código assembly específico da plataforma de execução.

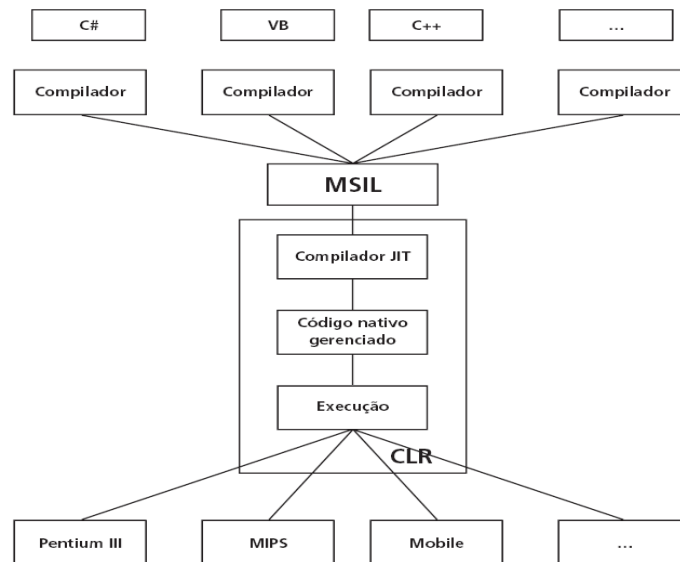


Figura 1.4

Gerenciamento da memória: GC (Garbage Collector)

O gerenciamento da memória é efetuado pelo runtime, permitindo que o desenvolvedor se concentre na resolução do seu problema específico. O que diz respeito ao sistema operacional, como o gerenciamento da memória, é feito pelo runtime. Como isso é efetuado? À medida que uma área de memória é necessária para alocar um objeto, o GC ou coletor de lixo (Garbage Collector) realizará essa tarefa, assim como a liberação de espaços de memória que não estiverem mais em uso. Para os que não trabalham com linguagens de programação como C ou C++, que permitem o acesso direto à memória *heap* via ponteiros, essa é uma das maiores dores de cabeça que os programadores sofrem, ora por fazer referência a espaços de memória que não foram alocados, ora porque estes espaços já foram liberados anteriormente; é exatamente esse tipo de erro que o coletor de lixo nos ajuda a evitar. O gerenciamento da memória, quando efetuado diretamente pelo programador, torna os programas mais eficientes em termos de desempenho, mas ao mesmo tempo o penaliza, obrigando-o a alocar e desalocar memória quando assim é requerido. A .NET permite que o programador faça esse gerenciamento também, o que é chamado de “unsafe code” (código inseguro); entretanto, por default, oGC é o encarregado dessa tarefa, e o contrário não é recomendado.

Onde podemos usar a .NET?

Como consequência do que foi dito acima, a .NET se adapta perfeitamente ao desenvolvimento do seguinte tipo de aplicações:

- Aplicações clientes de front end
- Aplicações de middleware: Web services, aplicações do lado servidor (ASP.NET, SOAP, Web Services e XML)
- Aplicações para internet: a .NET fornece bibliotecas especializadas para o desenvolvimento de aplicações para Internet suportando os protocolos mais comuns: FTP, SMTP, HTTP, SOAP etc.
- Aplicações gráficas: via a biblioteca GDI+, a .NET dá suporte completo a

esse tipo de aplicações.

- Acesso a bancos de dados via ADO.NET: ADO.NET é uma evolução da tecnologia ADO usada amplamente no desenvolvimento de sistemas para bancos de dados. Entretanto, novas características são encontradas nessa nova biblioteca, como manipulação de dados na aplicação cliente, como se esta estivesse sendo manipulada no servidor. Isso implica em aplicações *connectionless* (sem conexão) com vistas a não degradar o desempenho do servidor de banco de dados, quando este está servindo 17 milhares de conexões simultaneamente. Ideal para desenvolvimento de aplicações OLTP, não é?
- Aplicações multitarefa: a biblioteca System.Thread dá suporte ao desenvolvimento de aplicações multitarefa. E muito mais!

Ambientes de Desenvolvimento .NET

No momento da elaboração deste material, temos conhecimento da existência dos seguintes ambientes de desenvolvimento para .NET:

- *.NETSDK Framework 1.1, 2.0 e 3.0*: Este ambiente vem junto com o SDK .NET e é apenas de linha de comando. Pode ser baixado do site da Microsoft.
- *VISUAL STUDIO .NET (VS.NET 2003 e 2005)*: Este é um ambiente de desenvolvimento da mesma família das versões do Visual Studio da Microsoft, mas ele é completamente integrado com todas as linguagens às quais oferece suporte: C#, VB, Jscript e Managed C++. Ainda é possível estender o suporte do VS.NET para outras linguagens que não são nativas a ele. Neste livro, nos basearemos neste ambiente para a apresentação dos exemplos. Entendemos que a maioria dos desenvolvedores da plataforma Windows irá usá-lo e por isso achamos desnecessário usar outros ambientes de desenvolvimento.
- *C-SharpDevelop*: Este ambiente de desenvolvimento é da categoria Open Source, possui algumas funcionalidades de IDE, mas ainda está em fase de desenvolvimento e portanto ainda incompleto. Apesar de ser um bom produto e ser gratuito, não chega ao nível do VS.NET. O uso deste ambiente é encorajado pela iniciativa Open Source .NET cujo nome é MONO (<http://www.go-mono.com>), e cujo objetivo é migrar a .NET para o ambiente Linux.

As características do C#

Dentre as características essenciais do C# podemos citar:

- *Simplicidade*: os projetistas de C# costumam dizer que essa linguagem é tão poderosa quanto o C++ e tão simples quanto o Visual Basic.
- *Completamente orientada a objetos*: em C#, qualquer variável tem de fazer parte de uma classe.
- *Fortemente tipada*: isso ajudará a evitar erros por manipulação imprópria de tipos, atribuições incorretas etc.
- *Gera código gerenciado*: assim como o ambiente .NET é gerenciado, assim também o é C#.
- *Tudo é um objeto*: *System.Object* é a classe base de todo o sistema de tipos de C#.

- *Controle de versões:* cada assembly gerado, seja como EXE ou DLL, tem informação sobre a versão do código, permitindo a coexistência de dois assemblies homônimos, mas de versões diferentes no mesmo ambiente.
- *Suporte a código legado:* o C# pode interagir com código legado de objetos COM e DLLs escritas em uma linguagem não-gerenciada.
- *Flexibilidade:* se o desenvolvedor precisar usar ponteiros, o C# permite, mas ao custo de desenvolver código não-gerenciado, chamado “unsafe”.
- *Linguagem gerenciada:* os programas desenvolvidos em C# executam num ambiente gerenciado, o que significa que todo o gerenciamento de memória é feito pelo runtime via o GC (Garbage Collector), e não diretamente pelo programador, reduzindo as chances de cometer erros comuns a linguagens de programação onde o gerenciamento da memória é feito diretamente pelo programador.

“Olá Mundo”: A estrutura básica de uma aplicação C#

O pequeno trecho de código a seguir implementa o clássico programa “Olá mundo”:

```
using System;
class AppPontoNet
{
    static void Main( )
    {
        //escrevendo no console
        Console.WriteLine("Olá mundo em C#");
        Console.ReadLine( );
    }
}
```

O Cabeçalho do programa

A primeira linha do nosso programa, que escreve no console “Olá mundo em C#”, contém a informação do namespace System, que contém as classes primitivas necessárias para ter acesso ao console do ambiente .NET. Para incluir um namespace em C#, utilizamos a cláusula using seguida do nome do namespace.

A declaração de uma classe

O C# requer que toda a lógica do programa esteja contida em classes. Após a declaração da classe usando a palavra reservada class, temos o seu respectivo identificador. Para quem não está familiarizado com o conceito de classe, apenas adiantamos que uma classe é um tipo abstrato de dados que no paradigma de programação orientada a objetos é usado para representar objetos do mundo real. No exemplo acima, temos uma classe que contém apenas o método Main() e não recebe nenhum parâmetro.

O Método Main()

Todo programa C# deve ter uma classe que defina o método Main(), que deve ser declarado como estático usando o modificador static, que diz ao runtime que o método pode ser chamado sem que a classe seja instanciada. É através desse modificador que o runtime sabe qual será o ponto de entrada do programa no ambiente Win32, para poder passar o controle ao runtime .NET. O “M” maiúsculo do método Main é obrigatório, e seu valor de retorno void significa que o método não retorna nenhum valor quando é chamado.

Algumas variantes do método Main()

```
//Main recebe parâmetros na linha de comando via o array
//args
static void Main(string[ ] args)
{
    //corpo do método
}

//Main tem como valor de retorno um tipo int
static int Main( )
{
    //corpo do método
}
```

A forma do método Main() a ser usada vai depender dos seguintes fatores:

- O programa vai receber parâmetros na linha de comando? Então esses parâmetros serão armazenados no array **args**.
- Quando o programa é finalizado, é necessário retornar algum valor ao sistema? Então o valor de retorno será do tipo **int**.

Um programa escrito em C# pode ter mais de uma classe que implementa o método Main(). Nesse caso, deverá ser especificado em tempo de compilação em qual classe se encontra o método Main(), que deverá ser chamado pelo runtime quando a aplicação for executada.

Exemplo:

```
using System;

class class1
{
    static void Main( )
    {
        Console.WriteLine("Método Main( ) da classe 1");
    }
}

class class2
{
    static void Main( )
    {
        Console.WriteLine("Método Main( ) da classe 2");
    }
}
```

O resultado da compilação deste programa é:

```
Class1.cs(6): Program 'C:\My Documents\Visual Studio Projects\twoMainMet\obj\Debug\twoMainMet.exe' has more than one entry point defined: 'class1.Main( )' Class1.cs(15): Program 'C:\My Documents\Visual Studio Projects\twoMainMet\obj\Debug\twoMainMet.exe' has more than one entry point defined: 'class2.Main( )'
```

Dentro do ambiente de desenvolvimento VS.NET proceda da seguinte forma para resolver esse problema:

1. Clique no menu **Project** e selecione a opção **Properties**.
2. Clique na pasta **Common Properties**.

3. Clique na opção **General**.
4. Modifique a propriedade **Startup Object**, selecionando a classe que contém o método `Main()` que você deseja que seja chamado pelo Runtime quando a aplicação for executada.
5. Clique em **Ok** e compile a aplicação de novo.

Alguns últimos detalhes adicionais

- Blocos de código são agrupados entre chaves { }.
- Cada linha de código é separada por ponto-e-vírgula.
- Os comentários de linha simples começam com duas barras `//`. Comentários em bloco são feitos usando os terminadores `/*` (de início) e `*/` (de fim).

```
/*  
Este é um comentário de bloco  
Segue o mesmo estilo de C/C++  
*/
```

- O `C#` é sensível ao contexto, portanto `int` e `INT` são duas coisas diferentes. `int` é uma palavra reservada que é um alias do tipo `System.Int32`. `INT` poderia ser um identificador, entretanto não é recomendado usar como identificadores de variáveis o nome de um tipo ou palavra reservada como no exemplo citado.
- Sempre declare uma classe onde todos os aspectos inerentes à inicialização da aplicação serão implementados, e obviamente, que conterá o método `Main()` também. No decorrer deste livro seguiremos fielmente essa regra nos nossos exemplos.

Interagindo com o console

Toda linguagem de programação oferece meios de interagir com o console, para ler ou escrever na entrada (geralmente o teclado) e saída padrão (normalmente o vídeo em modo texto). Em `C#`, temos uma classe chamada `Console` no namespace `System`, a qual oferece uma série de métodos para interagir com a entrada e saída padrão. Vejamos alguns exemplos:

```
public class stdInOut  
{  
    static void Main( )  
    {  
        char c;  
        string str;  
        //Escreve no console sem retorno de carro  
        Console.Write("Digite seu nome: ");  
  
        //Lê uma string do console. <Enter> para concluir  
        str = Console.ReadLine( );  
  
        //Escreve no console sem retorno de carro  
        Console.Write("Digite uma vogal e tecla <Enter>:");  
  
        //Lê do console um caractere simples.  
        c = (char)Console.Read( );  
  
        //Escreve uma linha em branco  
        Console.WriteLine( );  
    }  
}
```

```

        //Escreve uma string no console
        Console.WriteLine("Seu nome é: {0}", str);

        //Escreve 1 caractere com ToString( ) para converter
        Console.WriteLine("Sua vogal: {0}", c.ToString( ));
        Console.ReadLine( );
    }
}

```

Como você pode ver no exemplo acima, para escrever no console usamos os métodos:

- *Console.Write()*, para escrever uma string sem retorno de carro;
- *Console.WriteLine()*, para escrever uma string com retorno de carro. Essa string pode ser parametrizada, o que significa que o conteúdo de variáveis pode ser mostrado no console. As variáveis a serem mostradas começam a partir do segundo parâmetro e são separadas por vírgula. Na string do primeiro parâmetro elas são representadas por números inteiros, a começar por zero, encerrados entre terminadores de início “{” e de fim “}”.

Exemplo:

```
Console.WriteLine("var1: {0}, var2: {1}, var3: {2}", var1, var2, var3);
```

Para ler dados da entrada padrão, usamos os seguintes métodos:

- *Read()*, para ler um caractere simples;
- *ReadLine()* para ler uma linha completa, conforme mostrado no exemplo acima.

Formatando a saída padrão

A formatação da saída padrão é feita usando os chamados “caracteres de escape” (veja a tabela abaixo). Vejamos um exemplo:

```
//\t = TAB
```

```
//\n = quebra de linha e retorno de carro (CR LF)
```

```
Console.WriteLine("var1: {0} \t var2: {1} \t var3: {2} \n", var1, var2, var3);
```

Caracter	Descrição
\n	Insere uma nova linha
\t	TAB
\a	Dispara o som de um alarme sonoro simples
\b	Apaga o caractere anterior da string que está sendo escrita no console (backspace)
\r	Insere um retorno de carro
\0	Caractere NULL (nulo)

Variáveis

Em C#, todas as variáveis são declaradas dentro do escopo de uma classe e podem ser dos seguintes tipos:

- **Locais**: são declaradas no escopo de um método, indexador ou evento e não possuem modificadores de acesso. A sua declaração se limita ao tipo seguido do identificador da variável.
- **Atributos de uma classe ou campos da classe**: a variável é declarada como membro de uma classe. A declaração deve ser efetuada como se segue:
[Modificador de acesso] [tipo atributo] <tipo da variável> <identificador>

Exemplo

```

public class App
{
    public int varInt;

    static void Main( )
    {
        int varLocal;
    }
}

```

O campo interno no exemplo acima está declarado com a variável varInt, já a variável local está declarada com a varLocal. O modificador de acesso, neste nosso exemplo é o **public**, o tipo do atributo é do tipo inteiro que no caso do C# declaramos como sendo do tipo **int**, como você pode perceber é bastante sutil a observação desse tipo de variáveis dentro do código C#. Os modificadores de acesso somente são usados para os membros de classe, e não para variáveis locais.

Os modificadores de acesso estão relacionados abaixo:

Modificador de Acesso	Descrição
<i>Private</i>	Este modificador só permite que a mesma classe acesse o membro de classe. Ou seja, quando declaramos uma variável com este modificador, somente a classe que possui este membro de classe é que poderá acessar esta variável.
<i>Protected</i>	Com este modificador de acesso, a classe que possui a variável compartilha a variável com outras classes que possam vir a descender da classe proprietária da variável.
<i>Public</i>	Este modificador de acesso permite que outras classes possam acessar esta variável.

O Sistema de Tipos em C#

Em C#, todo tipo é derivado da classe *System.Object*, que constitui o núcleo do sistema de tipos de C#. Entretanto, os projetistas da linguagem, e não apenas da linguagem, mas de .NET como um todo, sabem perfeitamente das implicações de ter um sistema de tipos onde tudo é um objeto: queda de desempenho. Para resolver este problema, eles organizaram o sistema de tipos de duas formas:

- **Tipos Valor**: variáveis deste tipo são alocadas na pilha e têm como classe base *System.ValueType*, que por sua vez deriva de *System.Object*.
- **Tipos Referência**: variáveis deste tipo são alocadas na memória heap e têm a classe *System.Object* como classe base.

Os tipos valor, que são os tipos primitivos, tais como: *int*, *float* e *char*, estes tipos não precisam ser alocados na memória heap, pois como o acesso seria muito custoso para o sistema da arquitetura do computador, depreciando a performance de acesso; como o acesso a esse tipo de dado necessita ser ágil, então estes tipos são alocados na pilha da memória para otimizar a performance de acesso.

Veja na figura a seguir a hierarquia de tipos em C# e .NET:

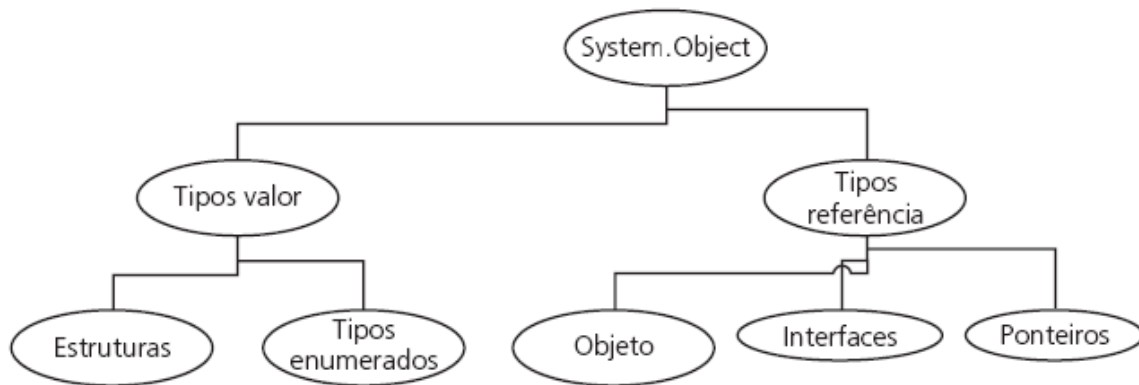


Figura 2.1

Tipos Valor

Tipos valor não podem ser usados como classes base para criar novos tipos porque estes são implementados usando classes chamadas “seladas”, a partir das quais não é possível implementar o mecanismo de herança. Antes de serem usados, os tipos valor devem ser inicializados, caso contrário o compilador acusará um erro.

Os tipos valor são subdivididos em duas categorias:

1. *Estruturas*
2. *Enumerados*

Estruturas

Estruturas são usadas para implementar tipos simples chamados de primitivos em outras linguagens de programação, são criadas na pilha e ainda oferecem muito do potencial de uma classe a um custo menor. Os seguintes tipos são implementados usando estruturas:

- **Tipos primitivos**
 - *Númericos*: inteiros, ponto flutuante e decimal
 - *Booleanos*: verdadeiro e falso
- **Tipos definidos pelo usuário**: estruturas propriamente ditas que permitem que o usuário crie seus próprios tipos.

Enumerados

São usados para implementar listas de valores constantes, os quais podem ser de qualquer tipo inteiro (long, int etc.); porém não podem ser do tipo char. Cada constante tem um valor inteiro associado, o qual pode ser sobrescrito quando assim definido na lista enumerada. Os valores inteiros associados a cada constante da lista enumerada começam a partir de zero. Abaixo segue um exemplo:

```
enum NovosTipos
{
    Valor1,
    Valor2,
    Valor3,
    Valor4,
```

```
} ...
```

Como já foi explicado, cada valor dentro da declaração **enum** corresponde a um valor **inteiro**, ou seja, podemos realizar um **cast** para podermos acessar o índice que o item selecionado representa em relação ao tipo inteiro, como mostra o exemplo abaixo:

```
int variavel = (int)NovosTipos.Valor1;
```

Neste exemplo estamos acessando o valor inteiro que Valor1, dentro do tipo enumerado representa, e estamos repassando este valor para a variável chamada variavel.

Classes

Uma classe é um tipo referência e podemos defini-la como a implementação de um tipo abstrato de dados que modela objetos do mundo real. Assim sendo, a classe é apenas uma representação de um objeto do mundo real dentro do mundo orientado a objetos. Uma classe define *atributos* e *métodos* que implementam a estrutura de dados e as suas operações, respectivamente.

Ao criarmos “variáveis” cujo tipo é uma classe, dizemos que estamos instanciando a classe, e por sua vez, instâncias de classes são chamadas de *objetos*. A declaração de uma classe em C# é como se segue:

```
[modificador de acesso] class <identificador> : [classe base]
{
    //declaração de atributos e métodos
}
```

Exemplo de declaração de classe:

```
public class MinhaClasse : System.Object
{
    private string nome;
    private float limiteCredito;
    private int id;
}
```

Em C# a implementação de propriedades e métodos é InLine, como segue o exemplo abaixo de implementação da propriedade:

```
public string Nome
{
    get { return nome; }
    set { nome = value; }
}
```

Membros de uma classe

Os membros de uma classe podem ser:

- Atributos
- Métodos
- Propriedades
- Eventos
- Constantes

- Indexers
- Operadores

Membros estáticos e membros instâncias

Os membros de uma classe podem ser classificados da seguinte maneira:

- ***Estáticos***: Só podem ser chamados usando o identificador da classe, mas não através das suas instâncias, o que significa que podem ser chamados sem ter de se criar uma instância da classe.
- ***Instâncias***: Os membros de instância não podem ser chamados sem antes ter instanciado a classe. Todos aqueles membros que não são precedidos do modificador **static** são entendidos como membros instâncias.

Os membros estáticos podem ser classificados da seguinte maneira:

- ***Atributos estáticos***: Independente do número de instâncias (objetos) que sejam criadas de uma classe, existe apenas uma única cópia de um atributo estático. Atributos estáticos só podem ser modificados na classe através de métodos estáticos. A tentativa de acessar um membro estático via uma instância gera um erro em tempo de compilação. Da mesma forma, atributos de instância só podem ser acessados via métodos de instância.

Exemplo

```
using System;

class Class1
{
    //atributo estático
    public static int total = 0;
    public int naoEstatico;
    //método estático

    public static void inc( )
    {
        total++;
    }
}

class App
{
    static public void Main( )
    {
        Console.WriteLine(Class1.total++);
        Class1.inc( );
        Class1 cl1 = new Class1( );
        Console.WriteLine(Class1.total++);
        Class1 cl2 = new Class1( );
        Console.WriteLine(Class1.total);
        Console.ReadLine( );
    }
}
```

Na saída do console temos os seguintes resultados ao compilarmos o nosso, exemplo:

```
0
2
3
```


Métodos

Os métodos são os equivalentes das funções ou procedimentos na programação estruturada. Entretanto, em C# os métodos estão diretamente ligados a uma classe e a sua declaração é feita da seguinte forma:

```
[modificador de acesso] [tipo do método] <tipo do valor de retorno>
<identificador do método>([lista de parâmetros])
{
    //implementação
}
```

Alguns detalhes importantes sobre a declaração de métodos:

- Os parâmetros consistem em uma lista de variáveis separadas por vírgula, cada um precedido pelo tipo mais o tipo da variável seguido do seu identificador.
- Um método pode simplesmente não receber parâmetros.
- Os métodos podem ter variáveis locais, as quais são declaradas na sua implementação.

Exemplo de método sem parâmetros:

```
public void metodoTeste( )
{
    int numero=0;
    numero++;
}
```

Exemplo de método com parâmetro:

```
public int metodoTesteComParametro(int param1, string param2)
{
    //aqui você pode utilizar os parâmetros do seu método da maneira que for
    necessário dentro de sua regra de negócio
    int variavel1 = param1;

    return <retornar qualquer valor inteiro dentro de sua regra de negócio>
}
```

Constantes

Constantes são atributos cujo valor é determinado em tempo de compilação. Exemplos de atributos constantes são: a variável matemática PI, a temperatura de ebulição da água (100°C) etc. Nesses casos, usamos uma constante porque o seu valor é conhecido em tempo de compilação e não mudará durante a execução do programa.

- Constantes devem ser declaradas dentro do escopo de uma classe ou método sendo, neste último caso, visíveis apenas no método (locais).
- As constantes são por definição estáticas e devem ser acessadas através do identificador da classe e não por suas instâncias. Apesar disso, constantes podem ser referenciadas por membros instância da classe, o que não é verdade para atributos estáticos convencionais.

Podemos declarar classes cuja única finalidade seja à declaração de constantes.

Como veremos mais adiante, não será necessário instanciá-la; para evitar que isso seja feito pelos usuários da classe, declararemos um construtor privado. Retomaremos esse assunto novamente quando falarmos de construtores.

Construtores de instâncias

Construtores de instâncias, ou simplesmente construtores, são métodos chamados automaticamente quando da instanciação de uma classe, os quais implementam as ações necessárias para sua inicialização. Sempre que uma classe é instanciada, um construtor conhecido como “default” é chamado.

Exemplo:

```
using System;

class Circulo
{
    private int x,y;
    public Circulo ( )
    {
        Console.WriteLine("Construtor Default");
    }
}

class CsharpApp
{
    static void Main( )
    {
        CsharpEx cl1 = new CsharpEx( );
    }
}
```

Vamos ilustrar alguns conceitos a partir de nosso exemplo:

- Um construtor sempre tem o nome da classe da qual é membro
- Construtores podem receber parâmetros
- Construtores sem parâmetros são chamados de *construtores “default”*, e sempre são chamados quando da instanciação da classe, no nosso exemplo, a seguinte declaração:

```
CsharpEx cl1 = new CsharpEx( )
```

...é de um construtor default.

Quando a classe é instanciada, o construtor default sempre será chamado, e para que determinadas ações sejam executadas a partir dele, este precisa ser modificado conforme fizemos no nosso exemplo.

- Construtores com parâmetros são chamados de *construtores customizados*
- Construtores não possuem valor de retorno.

A seguir, modificamos o exemplo anterior e adicionamos à classe um construtor customizado:

```
using System;

class Circulo
{
    private int x, y;
    public Circulo( )
```

```

    {
        Console.WriteLine("Construtor Default");
    }

    public Circulo(int a, int b)
    {
        x = a;
        y = b;
        Console.WriteLine("Construtor Customizado");
    }
}

class CsharpApp
{
    static void Main( )
    {
        Circulo cl1 = new Circulo( );
        Circulo cl2 = new Circulo(1,2);
        Console.ReadLine( );
    }
}

```

A saída no console deste programa é como se segue:

Construtor Default
Construtor Customizado

Destrutores

Destrutores são métodos membros de uma classe que são executados automaticamente quando um objeto é destruído. Algumas características importantes de um destrutor:

- É chamado automaticamente quando uma instância de uma classe não pode ser mais referenciada, e então é removida da memória pelo GC (Garbage Collector)
- Não recebe parâmetros

Estruturas

Estruturas são tipos abstratos de dados muito semelhantes a classes. A diferença mais sensível entre classes e estruturas reside no fato de que esta última não é alocada na memória heap, e os seus valores são diretamente contidos na estrutura; *o que equivale a dizer que estruturas são tipos valor e não referência*. As características mais importantes de uma estrutura são as seguintes:

- Não suportam mecanismo de herança
- São usadas para modelar estruturas de dados pequenas. Os tipos **int**, **double** e **bool** são implementados usando estruturas
- Estruturas implementam Interfaces, conceito que veremos mais adiante
- São tipos valor, portanto são alocadas na pilha e não na memória heap
- Cada variável do tipo estrutura (**struct**) contém uma cópia dos seus valores e, portanto, ao atribuímos uma variável **A** do tipo **struct** a outra, **B**, do mesmo tipo, o que estamos fazendo é uma cópia dos valores de **A** em **B**; portanto, qualquer modificação em **B** não alterará os valores de **A**.

Declarando estruturas

Veja no exemplo a seguir a declaração de uma estrutura:

```

using System;

struct Circulo
{
    private int x, y;
    private float diametro;

    public Circulo(int a, int b, float Diametro)
    {
        x = a;
        y = b;
        diametro = Diametro;
    }
}

class CsharpApp
{
    static void Main( )
    {
        Circulo cl1 = new Circulo(0,1, 10);
        Console.ReadLine( );
    }
}

```

A declaração da estrutura:

```

[modificador de acesso] struct <nome da estrutura> : [interface]
{
    //membros da estrutura
}

```

Como você pode observar na sintaxe da declaração da estrutura acima, semelhantemente a uma classe, estruturas podem implementar interfaces; no capítulo sobre programação orientada a objetos falaremos a respeito desse assunto. De forma similar a uma classe, uma estrutura pode possuir construtores, porém não possui destrutores e os construtores têm de ser customizados. A tentativa de declarar o construtor default gera um erro de compilação. Para criarmos uma variável de um tipo **struct**:

```

<tipo_struct> identificador = new <tipo_struct>[parâmetros do construtor]

```

Uma vez que é feita a declaração da estrutura, os valores default dos seus atributos são configurados e copiados diretamente na estrutura. Ao contrário das classes, que armazenam referências para áreas da memória, a estrutura contém os seus valores.

Estruturas de Controle Parte I

Declarações **if** permitem realizar avaliações de uma expressão e, dependendo da veracidade da avaliação, há a possibilidade de quebrar em seqüências lógicas específicas. C# fornece 3 formas de declarações **if**: **if** simples, **if-then-else** e **if-else-if-else**.

if Simples

```

if (Expressão Booleana)
[ { }
    <commando a serem executados com base na condição>
[ { }

```

Como esperado, a expressão booleana deve avaliar, também, true ou false. Quando a expressão booleana é verdadeira, o programa executa a condição, como segue exemplo abaixo:

```
if (args.Length == 1)
{
    Console.WriteLine("Conteúdo do argumento: {0}", args[0]);
}
```

if-then-else

O if simples, garante apenas que você possa executar ações de uma condição verdadeira. Isso pode ser feito ou não. Para gerenciar ambas as condições verdadeira e falsa, use a estrutura condicional if-then-else, ela tem a seguinte forma:

```
if (expressão booleana)
{
    <comandos para condição verdadeira>
}
else
{
    <comandos para condição falsa>
}
```

Exemplo:

```
if (args.Length == 0)
{
    Console.WriteLine("Conteúdo do argumento: {0}", args[0]);
}
else
{
    Console.WriteLine("Nao existe argumento");
}
```

if-else if-else

Algumas vezes isso é necessário para avaliar múltiplas condições para determinar qual ação será executada. Neste caso use a estrutura condicional if-else if-else. Aqui está sua forma normal:

```
if (Boolean expression)
{
    true condition statement(s)
}
else if (Boolean expression)
{
    true condition statement(s)
}
.
.
.
```

```
else if (Boolean expression)
{
    true condition statement(s)
}
else
{
    false condition statement(s)
}
```

Numa ordem sequencial, cada declaração começa com if e continua através de cada else if, essa avaliação continua até que uma de suas instruções avaliadas seja verdadeira ou até a última instrução condicional.

Exemplo:

```
if (args.Length == 0)
{
    Console.WriteLine("Primeira condição verdadeira");
}
else if (args.Length == 1)
{
    Console.WriteLine("Se a primeira não for verdadeira temos esta segunda opção verdadeira");
}
else
{
    Console.WriteLine("Senão esta será a opção falsa");
}
```

Switch

Quando há muitas condições para serem avaliadas, a declaração if-else if-else pode tornar-se complexa e difícil de manter. A solução mais limpa e clara para esta situação é a declaração Switch. O Switch permite testar qualquer valor inteiro ou string, como também múltiplos valores. Quando o teste produz o resultado esperado, todas as declarações associadas com aquele resultado serão executadas. Aqui está a forma básica do Switch:

```
switch(integral or string expression)
{
    case <literal-1>:
        statement(s)
        break;
    .
    .
    .
    case <literal-n>:
        statement(s)
        break;
    [default:
        statement(s)]
}
```

Estruturas de Controle Parte II

Loops são estruturas necessárias para executar múltiplas sequencias lógicas múltiplas vezes num programa. Em C# existem quatro tipos de estruturas de repetição: while, for, foreach e o do...while.

while

Esta estrutura de repetição é utilizada quando precisamos realizar a checagem antes mesmo de começar o processo de repetição do laço. Abaixo está a forma de declaração do while:

```
while (<Condição>)  
{  
    <comandos a serem executados>  
}
```

Exemplo de uso

```
string doAgain = "Y";  
int count = 0;  
string[] siteName = new string[10];  
  
while (doAgain == "Y")  
{  
    Console.WriteLine("Please Enter Site Name: ");  
    siteName[count++] = Console.ReadLine();  
  
    Console.WriteLine("Add Another?: ");  
    doAgain = Console.ReadLine();  
}
```

do..while

Esta estrutura de repetição é utilizada quando necessitamos que, antes mesmo da validação booleana, as instruções sejam executadas pelo menos uma vez. Abaixo segue a forma de declaração do do..while:

```
do {  
    Statement(s)  
} while (Boolean expression);
```

Exemplo de uso:

```
do  
{  
    Console.WriteLine("");  
    Console.WriteLine("A - Add Site");  
    Console.WriteLine("S - Sort List");  
    Console.WriteLine("R - Show Report\n");  
  
    Console.WriteLine("Q - Quit\n");  
}
```

```

Console.WriteLine("Please Choose (A/S/R/Q): ");

choice = Console.ReadLine();

switch (choice)
{
    case "a":
    case "A":
        Console.WriteLine("Add Site");
        break;
    case "s":
    case "S":
        Console.WriteLine("Sort List");
        break;
    case "r":
    case "R":
        Console.WriteLine("Show Report");
        break;
    case "q":
    case "Q":
        Console.WriteLine("GoodBye");
        break;
    default:
        Console.WriteLine("Huh??");
        break;
}

} while ((choice = choice.ToUpper()) != "Q");

```

For

Este tipo de loop (laço) são bastante utilizados quando sabemos previamente o número de repetições que serão necessárias para serem executadas. Abaixo segue a forma de declaração do for:

```

for (initializer; Boolean expression; modifier)
{
    statement(s)
}

```

A seção initializer somente será executada uma única vez, ou seja, a partir do momento que seja necessário entrar no loop e que o initializer tenha sido executado, apenas o boolean expression e o modifier serão executados, até que a expressão booleana seja satisfeita.

Exemplo de uso:

```

int n = siteName.Length-2;
int j, k;
string save;

```



```

for (k=n-1; k >= 0; k--)
{
    j = k + 1;
    save = siteName[k];
    siteName[n+1] = save;

    while ( String.Compare(save, siteName[j]) > 0 )
    {
        siteName[j-1] = siteName[j];
        j++;
    }
    siteName[j-1] = save;
}

```

foreach

O foreach é indicada quando precisamos trabalhar com interação com estruturas de coleção, onde não precisamos mais nos preocupar em incrementar o acesso ou gerir o acesso ao próximo item da coleção, sendo assim, basta que você apenas declare uma variável do tipo do item que está sendo armazenado dentro da coleção que a própria estrutura de repetição foreach é quem irá encarregar-se de gerenciar todo esse acesso. Abaixo segue a forma de declaração:

```

foreach (type identifier in collection)
{
    statement(s)
}

```

Exemplo de uso:

```

foreach(DataRow dr in ds.Tables[0].Rows)
{
    dr["<NOME_DO_CAMPO_NO_BANCO>"];
}

```

Arrays

Em C#, arrays são objetos cuja classe base é System.Array. Os arrays podem ser unidimensionais, multidimensionais ou ainda arrays de arrays, cujos elementos são outros arrays.

Declarando arrays

A seguir, temos a declaração de um array de inteiros unidimensional:

```
int[] arrInt = new int[2];
```

As três formas a seguir são válidas para inicializar um array quando é feita sua declaração:

```

int[] arrInt = new int[2] {0,1};
int[] arrInt = new int[] {0,1};
int[] arrInt = {0,1};

```

Declarando um exemplo multidimensional de 2x2 dimensões:

```
int[,] arrInt = new int[2,2];
```

Inicializando o array:

```
int[,] arrInt = new int[2,2] {{0,0},{0,1}}
```

Basicamente, o que estamos fazendo no último exemplo é declarar um array que, por sua vez, contém dois arrays bidimensionais. Veja um exemplo completo:

```
using System;

public class clArrays
{
    private int[] arrInt = new int[2] {1,2};
    private int[,] multInt = new int[2,2] {{1,2},{3,4}};
    private int[,] ArrDeArr = new int[2][,] {new int[2,2] {{1,2},{3,4}}, new int[2,2] {{5,6},{7,8}}};

    public void ImprimirArray( )
    {
        for (int i=0; i < arrInt.Length; i++)
        {
            Console.WriteLine("Elemento {0}: {1}", i, arrInt[i]);
        }
    }
}

class app
{
    static void Main( )
    {
        clArrays arrExemplo = new clArrays( );
        arrExemplo.ImprimirArray( );
        Console.ReadLine( );
    }
}
```

Preenchendo um array bidimensional:

```
public void preencherArrayBi( )
{
    for (int i=0; i < multInt.GetLength(0); i++)
        for (int j=0; j < multInt.GetLength(1); j++)
        {
            multInt[i,j] = i*j;
        }
}
```

No exemplo acima, usamos o método `GetLength()` da classe `System.Array` para saber o número de elementos de cada dimensão. Esse método recebe como parâmetro um número inteiro que corresponde à dimensão acerca da qual queremos conhecer o número de elementos. Preenchendo um array de arrays:

```
public void preencherJaggedArray( )
{
    for (int m=0; m < ArrDeArr.Length; m++)
        for (int i=0; i < ArrDeArr[m].GetLength(0); i++)
            for (int j=0; j < ArrDeArr[m].GetLength(1); j++)
```

```

        {
            ArrDeArr[m][i,j]= i+j;
        }
    }

```

Mostrando um array bidimensional no console:

```

public void ImprimirArrayBi( )
{
    Console.WriteLine("Array Bi-dimensional");

    for (int i=0; i< multInt.GetLength(0); i++)
    {
        for (int j=0; j < multInt.GetLength(1); j++)
        {
            Console.Write("{0}\t", multInt[i,j]);
        }

        Console.WriteLine(" ");
    }
}

```

Mostrando um array de arrays no console:

```

public void ImprimirJaggedArray( )
{
    Console.WriteLine("Imprimindo Array de Arrays");

    for (int m=0; m < ArrDeArr.Length; m++)
    {
        Console.WriteLine("ArrDeArr[{0}]", m );
        for (int i=0; i< ArrDeArr[m].GetLength(0); i++)
        {
            for (int j=0; j < ArrDeArr[m].GetLength(1); j++)
            {
                Console.Write("{0}\t", ArrDeArr[m][i,j]);
            }

            Console.WriteLine("");
        }
    }
}

```

A seguir, temos a classe app que chama cada um dos métodos:

```

class app
{
    static void Main( )
    {
        clArrays arrExemplo = new clArrays( );
        arrExemplo.ImprimirArrayBi( );
        arrExemplo.ImprimirJaggedArray( );
        Console.ReadLine( );
    }
}

```

Operações com Arrays

- **Rank**: Propriedade que retorna o número de dimensões de um array. Exemplo:
Result = multInt.Rank;
- **Length**: Propriedade que retorna o número total de elementos de todas as dimensões de um array. Result = multInt.Length; //Result será igual a 4.
- **GetLength**: Como já vimos acima, este método retorna o número total de elementos de uma dimensão específica do array. Recebe como parâmetro um número inteiro que corresponde ao número da dimensão da qual se deseja saber o total de elementos. A numeração das dimensões começa por zero. Veja o uso deste método no exemplo acima.
- **Reverse**: É um método estático cujo objetivo é inverter a ordem dos elementos do array. Essa inversão pode ser completa ou parcial, indicando o índice inicial e final para a inversão do array.

Exemplo:

```
int [] arr = new int[5]{1,2,3,4,5};
Array.Reverse(arr,1,2); //Invertendo o array parcialmente
Array.Reverse(arr); //Invertendo o array completamente
```

Sort: Ordena o array passado como parâmetro.

Exemplo:

```
int [] arr = new int[5]{1,3,5,2,0};
Array.Sort(arr);
```

Programação OOP

A programação orientada a objetos (OOP) veio para ficar, sem dúvida nenhuma. Ela permite que sistemas complexos sejam desenvolvidos com mais facilidade, tanto na implementação inicial quanto na manutenção.

O produto mais popular do Visual Studio até hoje tem sido o Visual Basic. Porém, reclamava-se muito da ausência de um suporte mais completo a todos os requisitos que caracterizavam uma linguagem de programação orientada a objetos (OOP). Com a arquitetura .NET, a Microsoft parece ter resolvido atender ao clamor da comunidade de desenvolvedores atacando em todos os flancos.

Vários melhoramentos foram feitos no Visual Basic, de forma que ele pode ser considerado agora como orientado a objetos; mas é em C#, sem dúvida alguma, que se terá acesso a uma linguagem de programação que implementa, de maneira simples e direta (sem as complicações do C++), todos os requisitos de uma linguagem OOP em conjunto com uma forte tipagem de dados (um dos pontos fracos do VB). Com isso, a programação torna-se mais sólida e muitos erros podem ser eliminados ainda em tempo de compilação.

Por que OOP existe?

Antes de continuarmos, vamos voltar um pouco no tempo e entender de onde vem a idéia por trás da programação orientada a objetos. O conceito predominante de programação antes de OOP era a chamada programação procedural. Consistia basicamente em dividir a tarefa de programação em pequenos blocos de código

chamados de procedimentos (procedures, em inglês), também conhecidos na época como sub-rotinas.

Em todos os casos, o que se fazia, basicamente, era escrever um trecho de código que manipulasse os valores de algumas variáveis e desse algum tipo de retorno. Exemplificando (código em português):

```
x = 0
Enquanto x < 10
x = x + 1
Fim Enquanto
```

O exemplo acima é muito simples, mas bastante ilustrativo. Mostra que existem dados (variáveis) e códigos que manipulam esses dados (estruturas de controle). Qual o inconveniente disso? Supondo que x fosse um valor que tivesse de ser exibido na tela, seria necessário acrescentar algum código que fizesse isso. Ou seja, x não era “capaz” de se “auto-exibir” na tela. O código completo seria:

```
x = 0
Enquanto x < 10
x = x + 1
Fim Enquanto
PosicionarCursor 0,0
Imprimir x
```

Se fosse feita outra alteração no valor de x, você teria de executar novamente os comandos de impressão para que o valor fosse atualizado na tela. O ponto a que queremos chegar é que dados e códigos eram concebidos como elementos separados. Havia, inclusive, uma definição que dizia: dados + código = programa.

Com a OOP, uma das idéias básicas era eliminar essa distância entre dados e código e fazer com que ambos ficassem mais interligados. Ambos seriam capazes de interagir de forma mais homogênea e autônoma.

Primeiramente, expandiu-se a idéia de procedimento para a idéia de classe. Uma classe permite que vários procedimentos e dados sejam armazenados dentro dela. Os procedimentos passaram a chamar-se *métodos* e os dados passaram a chamar-se *propriedades*. Mas não foi uma mera maquiagem e uma mudança de nome para a mesma coisa. De fato, o conceito de classe mudou radicalmente a visão da programação.

Uma classe define como um objeto deve funcionar. Fazendo uma analogia clássica, é como o projeto de uma casa: estabelece como as coisas têm de ser. A partir dessa planta, podem ser construídas várias casas idênticas. Isso é chamado de instância em OOP. Quando dizemos instanciar uma classe, significa colocar “no ar” um objeto baseado na descrição da classe.

Conceitos de encapsulamento, herança e polimorfismo

OK, objetos têm propriedades que podem ser manipuladas e gerar resultados visíveis e imediatos. Você provavelmente deve estar dizendo: “isso eu já sabia e não foi pra isso que comprei este livro; quero saber como criar meus próprios objetos”.

Então vamos primeiramente entender a mecânica de funcionamento dos objetos. Como já dissemos, ao alterar o valor de uma propriedade, códigos são executados de forma a produzir um resultado visível. No exemplo do botão, qualquer modificação nas propriedades Color ou Text produzirão efeitos imediatos. Porém, ao usar um objeto, você não “vê” o código nem precisa conhecê-lo para criar um botão. Isto é chamado de *encapsulamento*. Os códigos usados para alterar cores, títulos, forma e aspecto do botão ficam escondidos na implementação da classe.

Herança significa partir de algo já pronto e modificá-lo com o propósito de deixá-lo mais adequado a determinada finalidade. Supondo que você tem uma classe botão com propriedades básicas; você pode herdar aquelas características básicas e adicionar ainda mais funcionalidade. O detalhe é que, ao “derivar” uma classe a partir de outra, você não precisa conhecer o código da classe anterior. A sua nova classe não trará os códigos da classe pai. Eles continuarão encapsulados lá e você poderá usá-los se quiser. O dado importante é que você não precisa manipular os códigos da classe pai.

Polimorfismo é a capacidade que um objeto tem de comportar-se de formas diferentes, com base no seus descendentes, onde em um object você pode ter armazenado algum outro objeto que seja descendente desse object e por meio de cast você pode fazer com que esse object comporte-se da maneira do seu descendente.

Implementação prática dos conceitos

Vejamos agora, em termos de código, o que pode ser feito para criar classes em C#. Conforme já foi visto no Capítulo 2, C# é uma linguagem inteiramente orientada a objetos, de forma que tudo deve ser implementado dentro de classes. Não existem variáveis ou procedimentos “soltos”.

Apenas para relembrar, o esqueleto básico de uma classe deverá ter mais ou menos o seguinte aspecto:

```
escopo class NomeClasse
{
    //Propriedades
    escopo tipo nome;
    escopo tipo nome;

    //Construtores
    escopo NomeClasse
    {
        //Especificações
    }

    //Métodos
    escopo tipo NomeMétodo
    {
        //Especificações
    }
}
```

Evidentemente, esse exemplo está muito simplificado, mas traz a idéia básica: você tem de planejar quais serão as propriedades, os métodos, os construtores, destrutores e seus escopos e tipos. Exemplo:

```

public class Cadastro
{
    //Propriedades
    public string CPF;
    public string NomeCliente;

    //Construtores
    public Cadastro( )
    {
        MessageBox.Show( "Eu sou o construtor 'default!'" );
    }

    public Cadastro( string fCPF, string fNome )
    {
        //Comandos de inicialização
        this.CPF = fCPF;
        this.NomeCliente = fNome;
        MessageBox.Show( "Eu sou um construtor customizado e recebi " + fCPF + " e " + fNome +
" como parâmetros." );
    }

    //Métodos
    public bool Gravou( )
    {
        //Código para gravação
        return true;
    }
}

```

Em C#, a regra para definir um construtor é muito simples: basta criar um método cujo nome seja idêntico ao da classe. Toda vez que uma instância da classe for criada, o construtor será automaticamente disparado. Observe também que, no exemplo anterior, foram criados dois construtores para a mesma classe. Isso, em OOP, é chamado de *sobrecarga* (overload) de método. A palavra *sobrecarga* pode trazer a idéia de estresse ou excesso de trabalho, mas nesse caso é um elemento que nos ajuda e muito! Quando um método é “sobrecarregado”, o editor do Visual Studio dá uma indicação clara disso; mostrando que há outros métodos sobrecarregados por meio de tipos de assinaturas diferentes. O compilador reconhecerá automaticamente qual assinatura você está usando por meio dos tipos dos parâmetros que estão sendo utilizados no método.

Abaixo, segue um exemplo de sobrecarga de construtores:

```

{
    //Construtor 'default' é chamado
    Cadastro Ficha1 = new Cadastro( );
    //Construtor otimizado é chamado
    Cadastro Ficha2 = new Cadastro( "123", "De Oliveira Quatro" );
}

```

Criação de Propriedades

Vamos explorar agora um pouco mais as possibilidades de criação de propriedades. Os exemplos anteriores foram muito simples. Uma propriedade, é preciso entender em primeiro lugar, se parece muito com uma simples variável, mas é bem mais do que isso. Uma propriedade (também chamada de “atributo”, dependendo da forma como foi definida) é um valor ao qual é associado um método. Toda vez que for lido ou

gravado o valor da propriedade, métodos podem entrar em ação. Essas ações são definidas pelas palavras `get` e `set`. Na prática, pode-se dizer que uma propriedade é composta de três elementos: um campo (local onde armazena-se o valor, também chamado de atributo), um método de leitura (`get`) e um método de gravação (`set`). Veja o exemplo da nossa classe `Cadastro` reescrita dentro dessas premissas:

```
class Cadastro2
{
    protected string fCPF;

    public string CPF
    {
        set
        {
            fCPF = value;
        }

        get
        {
            return fCPF;
        }
    }
}
```

Como dissemos anteriormente, temos três elementos. As variáveis `fCPF` e `fNome` são os campos (fields) onde os valores são efetivamente armazenados. Observe que se trata de variáveis do tipo *protected*, o que significa que só serão vistas pelas classes descendentes (lembre-se do conceito de “*ser*”) e não serão manipuladas por outras classes (lembre-se do conceito de “*ter*”). O `set` e o `get`, como você pode notar, servem para recuperar e gravar novos valores. A palavra *value* é reservada no C# para receber o valor passado para a propriedade. Veja o exemplo:

```
InstanciaCadastro2.CPF = "123"; //Dispara o "set"
x = InstanciaCadastro2.CPF; //dispara o "get"
```

Tempo de vida dos objetos

Até agora, temos falado de criar objetos, mas também é igualmente importante saber como e quando destruí-los. Vimos que para criar uma nova instância de uma classe basta usar a palavra `new`:

```
x = new ClasseExemplo( )
```

O próprio *runtime* de .NET vai se encarregar de destruir o objeto quando ele não for mais necessário. A pergunta é: como ele sabe que um objeto não é mais necessário? Resposta simples e incompleta: o *runtime* libera a memória alocada pelo objeto quando não existe mais nenhuma referência ligada ao objeto dentro do contexto corrente.

Tratamento de exceções

O tradicional termo tratamento de erros é agora chamado de tratamento de exceções. E veremos que faz mais sentido. Em C#, o tratamento de exceções é feito de maneira muito elegante e simples. O C# adota um estilo relativamente comum em outras linguagens, que é o de tratar erros como objetos que encapsulam todas as informações que necessitamos para resolvê-lo. Essa idéia é válida para todo o ambiente .NET e foi

batizada como SEH (Structured Exception Handling). A idéia básica é mais ou menos a seguinte: todo objeto que representa uma exceção é derivado de System.Exception. Essa classe possui os seguintes membros:

Propriedade	Significado
<i>HelpLink</i>	Retorna uma URL para um arquivo de Help descrevendo o erro em detalhes
<i>Message</i>	Esta propriedade é somente leitura e descreve o erro
<i>Source</i>	Retorna o nome do objeto ou aplicação que gerou o erro
<i>StackTrace</i>	Esta propriedade é somente leitura e contém uma string que identifica a sequência de chamadas que disparou o erro
<i>InnerException</i>	Pode ser usada para preservar os detalhes do erro ao longo de uma série de exceções

Exceções podem ser disparadas pelos mais diversos elementos. Podem ser disparadas a partir de erros do sistema operacional, de erros de dispositivos, de inconsistências de dados. Também podemos criar nossas próprias exceções. Vamos observar um exemplo de como disparar uma exceção e como tratá-la com a nossa classe de Cadastramento e validação do CPF:

```
public class Cadastro
{
    //Propriedades
    public string CPF;
    public string NomeCliente;

    public Cadastro( string fCPF, string fNome )
    {
        //Inicialização
        CPF = fCPF;
        NomeCliente = fNome;
        if ( CPF.Length != 11 )
        {
            //THROW é a palavra chave para gerar a exceção
            throw new Exception( "CPF Inválido" );
        }
    }
}
```

Colocamos uma validação no construtor, de forma que se forem passados menos de 11 caracteres, o CPF é automaticamente considerado incorreto. Ao instanciar essa classe com um parâmetro inválido, a classe irá disparar a exceção e o resultado na tela deverá ser uma tela similar a esta:



Observe que um dos detalhes importantes está na sentença “unhandled exception”. Traduzindo: exceção não-tratada. A exceção foi disparada pela palavra-chave *throw* e é preciso que haja um bloco de código preparado para manipular a classe de erro retornada. O formato de tratamento de uma exceção é o seguinte:

```
try
{
    //Código sujeito a exceções
}
catch( TipoExcecao1 e )
{
    //Tratamento para exceção tipo 1
}
catch( TipoExcecao2 e )
{
    //Tratamento para exceção tipo 2
}
catch
{
    //Tratamento para qualquer tipo de exceção
}
finally
{
    //Trecho que deve sempre ser executado, havendo ou não exceção
}
```

A estrutura *try..catch* requer que exista pelo menos um bloco *catch* vazio. Os blocos de *finally* e *catch(exception)* são opcionais. Nossa classe Cadastro deveria ser instanciada da seguinte forma:

```
try
{
    Cadastro teste = new Cadastro( "123", "De Oliveira Quatro" );
}
catch
{
    MessageBox.Show( "Impossível instanciar o objeto Cadastro" );
}
```

Como você deve ter observado no exemplo, entretanto, não existe uma pista

clara de qual foi o motivo que realmente fez a instanciação do objeto. Para termos um controle mais efetivo, poderíamos usar o catch com uma exceção customizada. Vamos criar uma exceção que será usada especificamente para o caso de encontrar um CPF inválido e colocar isso dentro de um contexto com mais outras possibilidades de erro. O esboço do código ficaria desta forma:

```
//Criação das classes
public class eCPFInvalido : Exception{ }

public class Cadastro
{
    //Campos
    public string CPF;
    public string NomeCliente;

    public Cadastro( string fCPF, string fNome )
    {
        //Inicialização
        NomeCliente = fNome;

        if ( CPF.Length != 11 )
        {
            throw new eCPFInvalido( );
        }
        else
        {
            CPF = fCPF;
        }
    }
}

//Código que irá testar o tratamento de exceções
//Insira-o dentro do evento click de um botão,
//por exemplo
try
{
    int x;
    int z = 0;
    Cadastro teste = new Cadastro( "123", "De Oliveira Quatro" );

    x = 3 / z; //Teste pôr esta linha antes da anterior
}
catch( eCPFInvalido Erro ) //Experimente inverter a ordem dos catches
{
    MessageBox.Show( "CPF Inválido" );
}
catch( Exception Erro )
{
    MessageBox.Show( "Impossível instanciar o objeto Cadastro: " + Erro.Message );
}
finally
{
    MessageBox.Show( "Sempre executado, haja erro ou não." );
}
```

Observe uma série de detalhes pertinentes ao exemplo anterior:

- O *catch(Exception Erro)* é o trecho que captura toda e qualquer exceção. Ele é praticamente igual a um *catch* vazio conforme demonstrado antes, por uma razão

muito simples: *Exception* é pai de todas as exceções. Qualquer exceção sempre se encaixará em *Exception* por causa disso. A razão pela qual usamos *Exception* em vez de um *catch* vazio foi a intenção de exibir a mensagem enviada pelo sistema na ocorrência do erro.

- O bloco *finally* é executado sempre, haja erro ou não. Neste caso, ele é executado DEPOIS do bloco *catch* que capturar a exceção. Se você tem o costume de programar em Delphi, vai notar que essa ordem é invertida em C#.
- Qualquer erro ocorrido dentro um bloco *try* desvia o fluxo de execução para o *catch* correspondente e NÃO retorna o fluxo de execução para o ponto que originou o erro ou a linha seguinte. Em outras palavras, não existe nada semelhante ao *Resume*, tão popular no VB, por exemplo.
- A ordem dos fatores altera (e muito) os resultados. Se você tentar, por exemplo, as inversões sugeridas pelos comentários no exemplo anterior, vai perceber que, no caso de alterar a ordem dos *catches*, você receberá um erro do compilador: “*A previous catch clause already catches all exceptions of this or a super type ('System.Exception')*” (Uma cláusula *catch* anterior já captura todas as exceções deste tipo ou de um tipo superior “*System.Exception*”). No caso de inverter as linhas de código, você vai fazer com que diferentes *catches* sejam executados.
- Apenas um- e somente um - bloco *catch* é executado para cada tratamento de exceção (*try*).

ADO.NET – Para Iniciantes

Namespaces de ADO.NET

Assim como tudo o mais em .NET, existem vários *namespaces* para a nova implementação ADO. Vejamos quais são:

<i>Nome</i>	<i>Finalidade</i>
<i>System.Data</i>	<i>É o núcleo do ADO.NET. Ali estão definidos os tipos que representam tabelas, linhas, colunas, restrições e conjuntos de dados. Este namespace não define conexões com nenhuma base, apenas a representação dos dados.</i>
<i>System.Data.Common</i>	<i>Contém os tipos compartilhados entre os provedores gerenciados. Muitos desses tipos funcionam como base para os tipos concretos definidos em OleDb e SqlClient.</i>
<i>System.Data.OleDb</i>	<i>Define os tipos que permitem se conectar a um provedor OleDb (ADO Clássico), enviar comandos em SQL e recuperar conjuntos de dados. As classes deste namespace são parecidas (embora não idênticas) às do ADO clássico.</i>
<i>System.Data.SqlClient</i>	<i>Este namespace define as classes que se comunicam diretamente com o Microsoft SQL Server, evitando passar pelo OleDb.</i>
<i>System.Data.SqlTypes</i>	<i>Representam os tipos de dados nativos do SQL Server. Embora você possa sempre usar os tipos genéricos, os deste namespace são otimizados especificamente para o SQL Server.</i>

Não tem momento exato para começar, mas quando se inicia um estudo, é importante partir do princípio de que tem que buscar o básico e então ir avançando até a evolução no conhecimento. Existem hoje muitas revistas, websites, e autores de altíssima qualidade aqui no Brasil. Mas sempre percebo que falta material para quem está começando. É uma tendência natural sempre surgirem novos artigos tratando de assuntos cada vez mais avançados ficando o novo público perdido, pois não encontra artigos básicos. Recebi diversos e-mails me cobrando um artigo como esse. Então resolvi tomar como meta direcionar vários artigos a esse propósito.

O requisito para este artigo é que você conheça o básico de banco de dados começando no MS ACCESS e partindo para o MS SQL Server 2000.x ou MSDE. Utilizaremos a partir deste ponto a sigla SGDB para nos referirmos ao servidor de banco de dados.

Introdução a Banco de Dados

O banco de dados sempre foi um ponto forte na aplicação. Sendo que uma falha na modelagem (preparação) pode prejudicar o desempenho da própria aplicação. Portanto é

fundamental seu conhecimento para que possa preparar da melhor forma possível, tal como dimensionar a solução a ser adotada. Dentre as opções disponíveis no mercado vamos rever as seguintes:

- **MS ACCESS** - Este é um ótimo banco de dados principalmente para quem está começando pela facilidade de utilização, você rapidamente aprende como utilizá-lo. Seu uso é recomendado apenas como doméstico e é uma ótimo caminho para, depois que se aprender, passar a utilizar o MS SQL Server

- **MS SQL Server** - Este é um servidor de banco de dados profissional, e é atualmente um dos melhores do mercado. Um grande destaque nesse produto é sua facilidade de operação, resultando em um ótimo produto e um custo mais baixo de manutenção, tal como custo de aquisição se comparado a produtos similares no mercado. Pode ser implementado de forma individual ou em paralelo (Cluster) com suporte a tolerância de falhas, desde é claro que possua hardware e software para isso.

- **MS MSDE** - Este é uma versão gratuita e similar ao SQL Server e tem com objetivo você desenvolver sua aplicação e testar sem a necessidade de ter o SQL Server. Para evitar o uso indevido essa versão conta com recurso que resulta em perda de desempenho no caso de mais de 5 transações simultâneas tal como limitação do tamanho do arquivo de dados. Eu recomendo seu.

Em todo artigo utilizaremos o MS SQL Server , por ser um servidor de banco de dados, ele armazena dentro do mesmo vários bancos de dados conforme a **Figura 1**, ficando cada aplicação utilizando um ou vários desses bancos, bastando é claro que tenha direito de acesso para isso. Todos nossos exemplos será no Database **Northwind**. Sendo este um banco de dados padrão que vem junto com o SQL Server já pra ser utilizado para testes pois já possui dados.

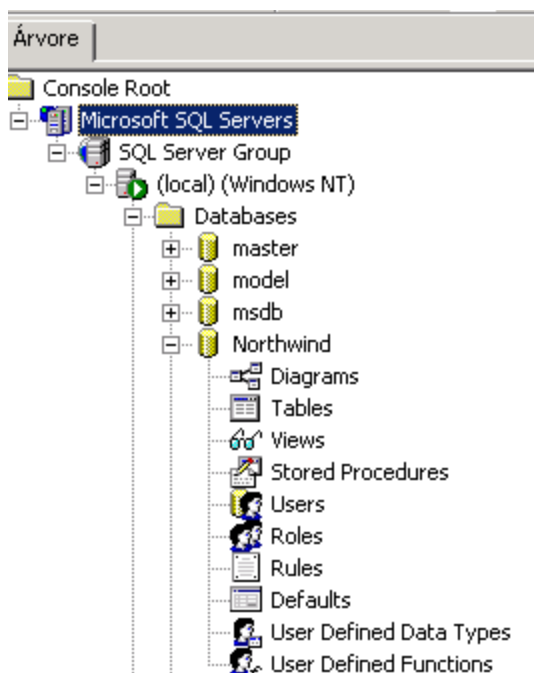


Figura 01 - Estrutura do MS SQL Server

Dentro de cada banco de dados temos as Tabelas (Tables) conforme **Figura 2** e dentro das tabelas temos as colunas (Column) aonde definimos que tipo de dados vamos armazenar conforme **Figura 3**.

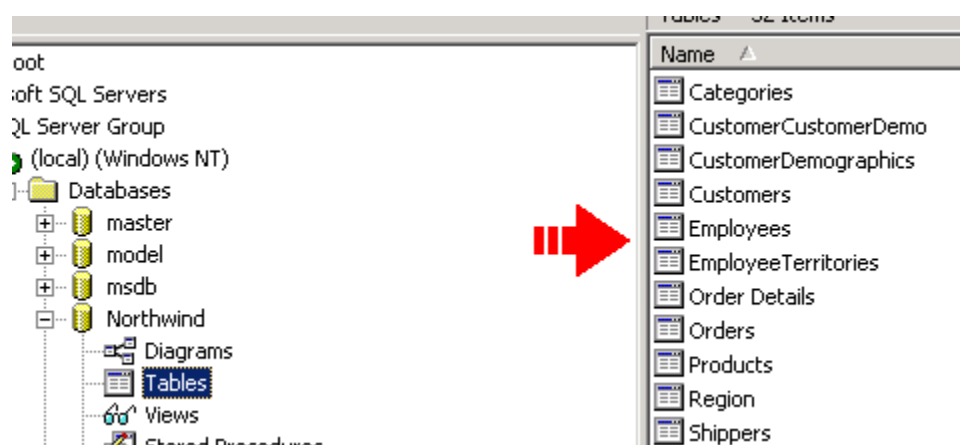


Figura 02 - Demonstrando as Tabelas (Tables).

	Column Name	Data Type	Length	Allow Nulls
	ProductID	int	4	
	ProductName	nvarchar	40	
	SupplierID	int	4	✓
	CategoryID	int	4	✓
	QuantityPerUnit	nvarchar	20	✓
	UnitPrice	money	8	✓
	UnitsInStock	smallint	2	✓
	UnitsOnOrder	smallint	2	✓
	ReorderLevel	smallint	2	✓
	Discontinued	bit	1	

Figura 03 - Estrutura da tabela e colunas (Column).

Conforme observado na **Figura 03**, a Coluna ProductID vai armazenar dados do tipo Inteiro e não vai aceitar nulo ou seja, essa coluna é de preenchimento obrigatório.

A maioria dos servidores de banco de dados possui suporte a Transact-SQL que é um protocolo padrão entre os SGDB para manipulação dos dados. O MS SQL Server oferece o utilitário chamado SQL Query Analyzer que é um console para utilização dos comandos SQL conforme **Figura 4**, está realizando uma consulta por meio do comando **Select**.

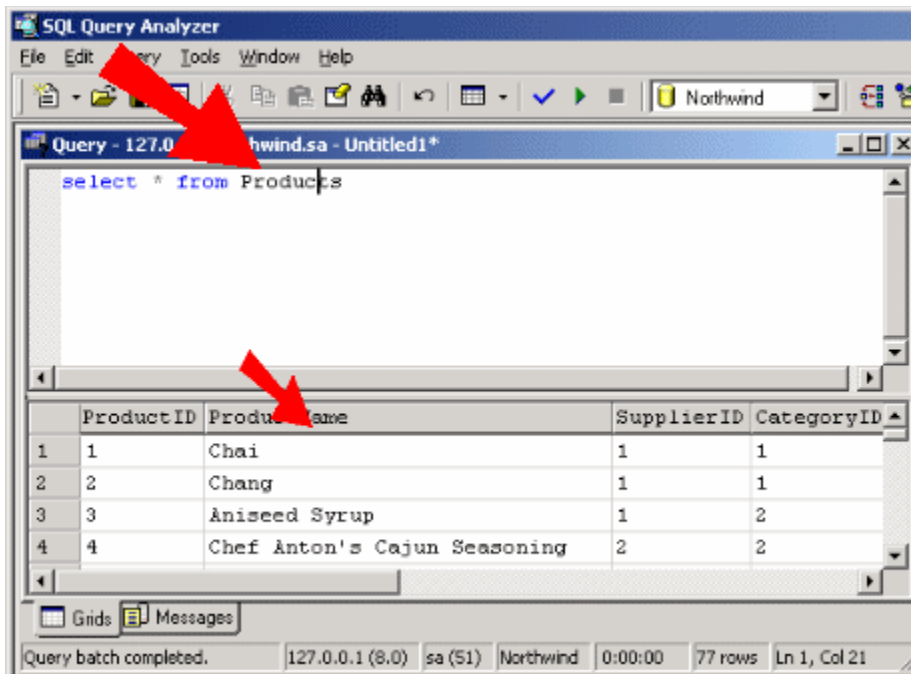


Figura 04 - Consultando dados usando Query Analyzer.

Principais comandos SQL:

- **Select** (Selecionar registros)
- **Delete** (Excluir registros)
- **Update** (Atualizar registros)
- **Insert** (Inserir registros)

-> **Where** (Utilizando em conjunto com select, update, delete para impôr uma condição para processar o comando)

Exemplos:

Select * from Products where ProductID = 1 (Seleciona todos registros da tabela produtos cuja coluna ProductID tenha valor igual a 1)

Delete Products where ProductID = 1 (Exclui todos registros da tabela produtos cuja coluna ProductID tenha valor igual a 1)

Update Products set ProductName = 'Chair' where ProductID = 1 (Atualiza todos registros na tabela produtos, coluna ProductName para 'Chair' cuja coluna ProductID=1)

insert into Products
(ProductName, SupplierID, CategoryID, QuantityPerUnit)
values
('Telefone',1,1,1) (Inclui uma linha na tabela Products)

Concluída essa introdução pesquise mais detalhadamente sobre o funcionamento do banco de dados e outras facilidades oferecidas como relacionamento entre tabelas,

integridade, constraints, procedures, views, triggers, backup e como gerar o diagrama do mesmo.

Histórico do acesso aos Dados.

Logo no início, para se ter acesso às informações do banco de dados era necessário ter vasto conhecimento das API de comunicação, e o programador acabava implementando seu código exclusivamente para cada versão de driver como a DBLIB (SQL Server), dedicando com isso muito tempo a este tipo de implementação. Com o crescimento do mercado e a persistência desse problema foi criado em 1990 com apoio da Microsoft e um consórcio de empresas, o padrão **ODBC** (Open DataBase Connectivity). Que é uma camada intermediária encarregada de cuidar da comunicação com a API deixando para o programador uma interface única e padrão para todo acesso ao SGDB confira na **Figura 4a**.

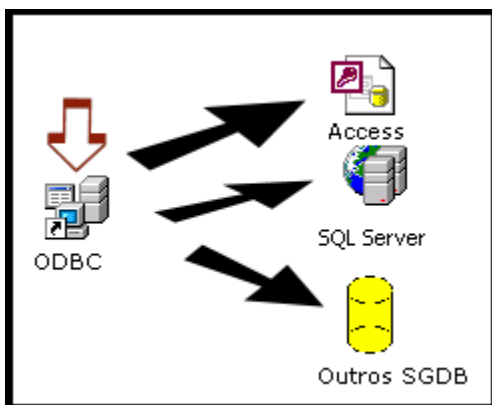


Figura 4a - Demonstrando ODBC

A partir do sucesso do ODCB e da experiência já estabelecida e necessidade de evolução foram surgindo outras propostas como o **DAO** (Data Access Objects) focado no MS ACCESS, sendo substituído logo depois pelo **RDO** (Remote Data Objects), uma vez que o DAO era realmente lento em conjunto com o ODBC.

O grande avanço da época deu-se em torno do **OLEDDB** que se assemelhou muito com a arquitetura do ODCB porém trouxe a implementação de interfaces COM e a estratégia da Microsoft UDA (Universal Data Access) com objetivo de armazenamento distribuído, como desde planilha até e-mails. Semelhante ao ODBC, o **OLEDDB** também foi sucesso sendo aderido até por banco de dados de padrão aberto. Para facilitar sua utilização foi criado o **ADO** (Activex Data Objects) com objetivo de consumir os recursos oferecidos pelos **OLEDDB**.

O que podemos concluir desse breve histórico é que foram criadas várias camadas de acesso ao banco de dados, com objetivo de simplificar e padronizar sua utilização. Isso realmente foi fundamental para o desenvolvimento e evolução das aplicações deixando o padrão **OLEDDB** em conjunto com o **ADO** na liderança no acesso a dados devido a um melhor desempenho no acesso a dados. Sendo utilizando por diversas soluções ficando o padrão **ODBC** para as soluções não compatíveis com **ODBC**. Sendo assim vale ressaltar que mesmo usando **OLEDDB** ou **ODBC** a aplicação vai ter uma perda de desempenho pois vai ter um camada intermediária entre sua aplicação e as APIs de acesso a dados.

O que é ADO.NET?

Já faz muito tempo que a Microsoft vem investindo em criar uma interface amigável de acesso a dados para as aplicações de forma a se obter os melhores recursos, chamada anteriormente de ADO, mecanismo esse que ainda é utilizado por diversas aplicações inclusive de outros fornecedores ver modelo na **Figura 4b**.

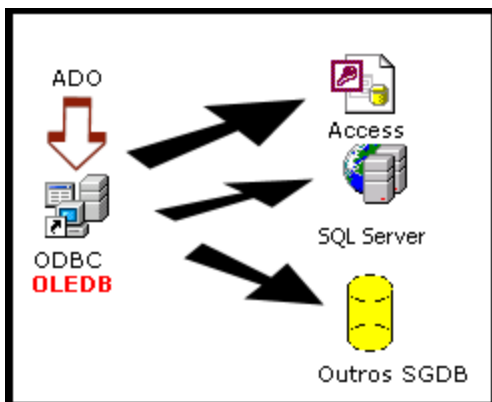


Figura 4b - Modelo ADO

O **ADO.NET** é uma completa reformulação desse mecanismo de acesso a dados, sendo uma nova geração dessa arquitetura completamente integrado ao .NET Framework oferecendo um vasto número de classes resultando numa fácil e eficiente comunicação com o SGDB permitindo todas operações necessárias.

Por ser integrado ao .NET tem o mesmo suporte a OOP, Compilação, Linguagens, Gerenciado, Coletor de lixo e **principalmente acesso nativo ao banco de dados sem intermediários "OLEDB" ou "ODBC"**. Confira na **Figura 4c**.

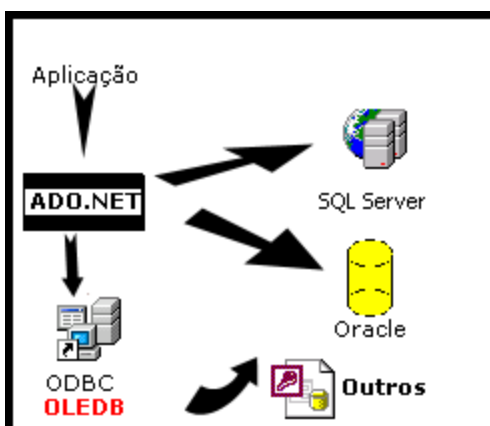


Figura 4c - Modelo ADO.NET com acesso nativo

Agora o acesso tornou-se mais otimizado e desconectado. Foi criado um repositório baseado em XML para que permita uma maior integração de dados entre os mais diversos sistemas de armazenamentos diferentes.

Para uma melhor compreensão vamos imaginar um cenário em que a geladeira de sua casa é o ADO.NET e o supermercado é o repositório de dados SGDB conforme **Figura 5**.

Portanto quando você vai ao supermercado e compra melão, cenoura e beterraba, coloca na sua sacola e traz pra casa, está neste momento abrindo uma conexão com banco, efetuando uma consulta usando Transact-SQL e recuperando os dados. Dando seqüência ao nosso cenário você normalmente pegaria essas compras (melão, cenoura e beterraba) e distribuíria pela geladeira em sua casa. Este seria apenas um modelo estabelecido pra melhor organizar, mas para o ADO.NET é diferente. Você no momento de armazenar não iria conseguir colocar melão e frutas na mesma localização, pois seria de tipos diferentes. Ou seja frutas é diferente de verduras, implicando assim que não são do mesmo tipo. Dando seqüência sabemos que nossa geladeira é ADO.NET por isso já temos na tela a quantidade de cada itens: Frutas/Verduras para lhe oferecer acesso com facilidade a todos itens, sem ter que abrir e sair procurando. Fazendo isso estaríamos exemplificando o que seria o DataSet que veremos posteriormente.

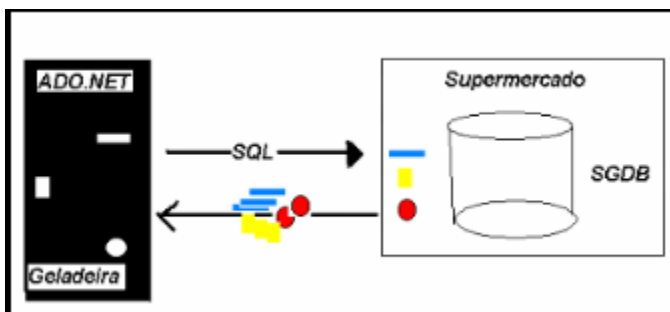


Figura 5 - Comparando ADO.NET

Ao longo desse artigo estaremos apresentando muitas novidades e a primeira delas será o acesso nativo a SGDB.

Conforme apresentamos no histórico sobre acesso a banco de dados, ao longo do tempo teve-se uma necessidade de se criar uma camada para padronizar o acesso, dentre as diversas tentativas prevaleceu o ODBC e OLEDB sendo inclusive suportados pelo novo ADO.NET, porém o mesmo agora dispõem de meios para acessar o SGDB de forma nativa sem a necessidade de intermediários já citados. Resultando em um acesso muito mais rápido como se estivesse implementando diretamente as APIs nativas do SGDB. As principais classes estão organizadas nos Namespaces (Espaço de nomes, destinado a organizar classes) abaixo:

- **System.data.SqlClient** (SQL Server 7.x ou superior) (Nativo)
- **System.data.OracleClient** (Oracle) (Nativo)
- **System.data.OleDb** (Access e todos outros banco de dados do mesmo padrão)
- **System.data.ODBC** (Todos bancos do padrão)

Portanto conforme **Figura 6** a partir de agora sempre que for utilizar SQL Server 7.x dê preferência para utilizar as classes **SQLClient** e usar de todos benefícios das classes nativas.

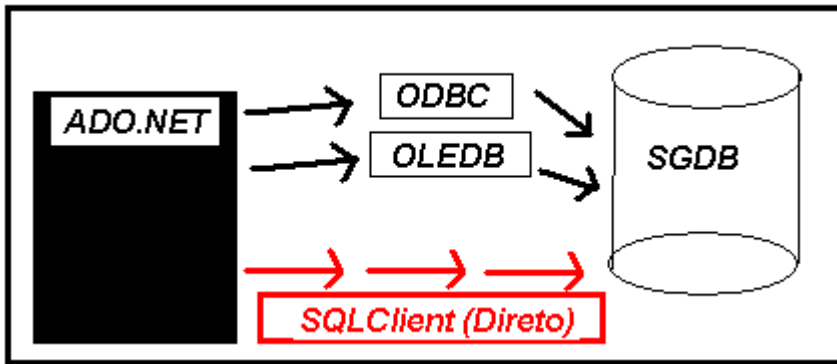


Figura 6 - Acesso nativo ao SGDB (Alto desempenho)

Compreendida as diferenças entre acesso nativo e acesso usando intermediários poderemos dividir o acesso do **ADO.NET** em duas partes Conectado (**DataReader**) e Desconectado (**DataSet**). Vamos comparar as diferenças entre ambos na **Tabela 01**:

Tabela 01 - Comparativo DataSet x DataReader

DataSet	DataReader
- Desconectado	- Conectado, Precisa ser aberto/fechado
- Leitura /Escrita	- Leitura
- Baseado em coleção, permite navegação para frente e para trás	- Apenas num sentido sem retorno (Forward only)
- Permite fazer bind para vários controles Xml	- Não usa cursor
- Permite ser serializado e utilizado em webservices.	- Acesso rápido
- Permite ser criado de forma automática usando visual studio .net Tools (arrastando pelo Server Explorer) ou programática via código	- Semelhante ao velho recordset
- Acesso mais lento	- Somente via forma programática
- Inclui varias tabelas, relacionamento, chave primeira...	- So faz bind de um controle

Observaremos um detalhamento nas **Tabelas 02 e 03**.

Tabela 02 - Detalhamento DataReader (Conectado)

DataReader (Conectado)	
Connection	Fornecesse o acesso ao banco de dados, nessa classe que você informa os dados de acesso ao banco de dados como usuário e senha. "ConnString"
Command	Nesta classe você fornece a query sql (select,delete,insert,update, procedure) para processar no banco de dados
DataReader	Faz a leitura dos registros no banco de dados.

Tabela 03 - Detalhamento DataSet DataSet (Desconectado)

DataSet (Desconectado)	
Connection	Fornecesse o acesso ao banco de dados, nessa classe que você informa os dados de acesso ao banco de dados como usuário e senha. "ConnString"
Command	Nesta classe você fornece a query sql (select , procedure) para processar no banco dedados
DataAdapter	Faz a leitura do banco de dados, extrai todos os dados de acordo com o command e preenche o DataSet
DataSet	Repositório de dados baseado em XML que pode ser transportado pelos webservices.

Trataremos com maiores detalhes sobre DataSet e sobre DataReader nos próximos tópicos, neste momento observe na **Figura 07**, a representação das classes apresentadas na **Tabelas 01, 02 e 03**.

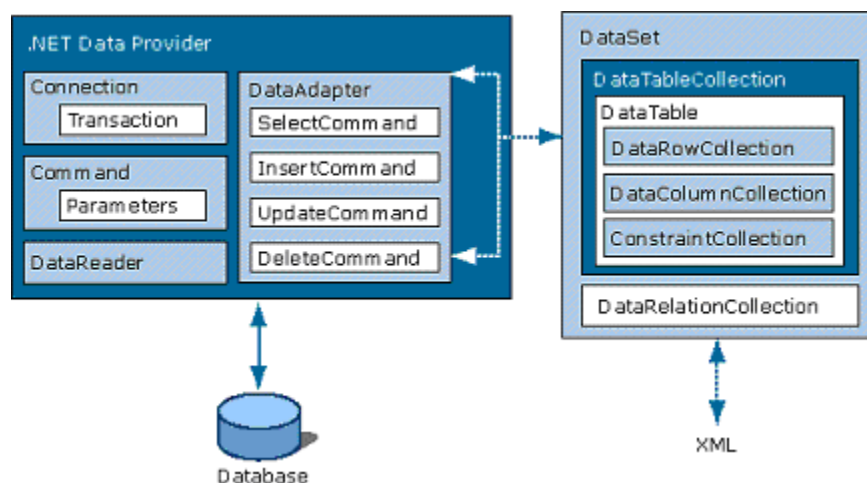


Figura 07 - Modelo de acesso a dados (DataReader e DataSet)

A classe **Connection** é utilizada em qualquer tipo de acesso ao SGBD sendo ele conectado ou não e a mesma recebe um parâmetro muito importante de acordo com o tipo de banco de dados que você vai utilizar. Tal como se você estivesse utilizando OLEDB vai passar um parâmetro, se for ODBC vai passar outro. Veja alguns exemplos na **Tabela 04**.

Tabela 04 - Exemplos de string de conexão.

Connection	Exemplo
SQLClient	Data Source=localhost; User ID=sa; Password=;Initial Catalog=Northwind;
OleDb (SqlServer)	Provider=SQLOLEDB.1;Data Source=localhost;Initial Catalog=Northwind;User ID=sa;Password=;
OleDb (Access)	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\db\ northwind.mdb;Persist Security Info=False;

Lembre sempre de verificar se sua string de conexão é a adequada para o seu SGBD e para classe do ADO.NET que você está utilizando. Sendo que sua preferência será primeiro classes nativas (SQLClient, OracleClient) depois OleDb e por último ODBC.

Outra grande novidade é a melhoria no suporte a transações durante o processo de comunicação com o SGBD. Transações é a capacidade de retornar as alterações feitas caso tenha algum problema. Isso é muito comum de se implementar dentro do banco de dados, agora ficou muito fácil de se implementar também no código, inclusive com suporte a Save Point, ou seja, retorna a um determinado ponto caso tenha alguma falha processando o Transact-SQL.

DataBind

Este agora é um termo muito utilizado no .NET pois é a capacidade do controle de automaticamente ler um DataReader ou DataSet e exibir os dados na tela, seja um ListBox, DataGrid, DropDown...

Utilizando deste recurso o programador não precisa mais fazer loop no retorno do banco de dados para popular estes controles. A única atenção que você precisa ter é que um DataReader só pode ser lido uma vez. Por isso ele somente carrega um controle. Já o DataSet por ser desconectado pode ser lido quantas vezes for necessárias e pode carregar mais de um controle.

As principais propriedades encontradas nos controles para usar o recurso DataBind são:

- **DataSource** (Define fonte de dados)
- **DataBind** (Aciona a leitura da fonte de dados)
- **DatavalueField** (Define valor para ListBox, Dropdown)
- **DatatextField** (Define texto para ListBox, Dropdown que será apresentado na tela)

As propriedades `DataValueField` e `DataTextField` devem ser preenchidas com o nome da coluna do banco de dados.

Utilizando o `DataReader`

É a forma de acesso mais rápido à base de dados e tem um consumo muito baixo de memória, pois dispõe de menos recursos em cima dos dados retornados, uma vez que só vai numa direção, sem retorno. Porém como já vimos só consegue utilizar o `DataReader` para carregar apenas um controle.

Para cada classe de acesso você usa também um `DataReader` específico. Por exemplo, para classe `SQLClient`, usa-se o `SqlDataReader` e para `OleDb` usa-se `OleDbDataReader` e assim por diante pois cada classe tem implementações específicas.

O `DataReader` se assemelha muito ao ADO você precisa explicitamente abrir e fechar a conexão (`Connection`) com o SGDB. Conforme **Figura 07** pra utilizar essa classe você vai precisar dos seguintes passos.

1. **Connection** - Abre e fecha conexão, necessita de uma string de acesso conforme **Tabela 04**.
2. **Command** - Vai processar seu comando `Transaction-SQL` no SGDB.
3. **DataReader** - Faz a leitura do retorno

Até este ponto você deve estar achando muito fácil acessar o banco de dados e realmente é muito simples mesmo, não tem por que ser complicado. Outra questão importante é que o `DataReader` somente lê o retorno do banco de dados (`Read Only`). Mas dependendo do comando SQL que você esteja utilizando você conseguirá excluir, alterar, inserir. O termo somente leitura é após o processamento do SQL no SGDB, ou seja depois que o `Command` já processou e o `DataReader` está recebendo a resposta.

Como o `DataReader` do ADO.NET é muito parecido com o ADO que utilizávamos antigamente vamos observar o código abaixo e depois criaremos um exemplo similar.

```
<%  
'Arquivo: Teste.asp  
set conn = server.createobject("adodb.connection")  
Conn.Open "Provider=SQLOLEDB.1;Data Source=localhost;Initial  
Catalog=Northwind;User ID=sa;Password=;"  
sql="select * from Products"  
set rs=Conn.Execute(SQL)  
do while not rs.eof  
response.write cstr(rs.fields("ProductID"))+"<BR>"  
response.write cstr(rs.fields("ProductName"))+"<hr>"  
rs.movenext  
loop  
Conn.Close  
rs.Close  
set rs=nothing  
set conn=nothing
```

%>

Esse código é de uma página teste.asp que está acessando SGDB SQL Server utilizando OleDb, vamos agora criar um código similar utilizando ADO.NET e como vamos usar SQL Server já vamos montar o exemplo usando a classe SQLClient. Para começar adicione uma nova página ao seu projeto, no solution explorer botão direito>Add Web Form e coloque o nome teste.aspx, feito isso e já tendo a página carregada efetue dois cliques na mesma para ir para parte do código e modifique conforme conteúdo da **Listagem 01**.

Listagem 01- Código para utilizar SqlDataReader

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.OleDb

Public Class teste
    Inherits System.Web.UI.Page
    Private Sub Page_Load _
        (ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

        Dim conn As New SqlConnection 'Criando objeto Connection
        Dim cmd As New SqlCommand 'Criando objeto Command
        Dim dr As SqlDataReader 'Criando objeto DataReader

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"

        cmd.Connection = conn
        cmd.CommandText = "Select * from Products"
        cmd.CommandType = CommandType.Text

        conn.Open() 'Abrindo Conexão

        dr = cmd.ExecuteReader

        Do While dr.Read

            Response.Write(Convert.ToString(dr("ProductID")) + "<BR>")
            Response.Write(Convert.ToString(dr("ProductName")) + "<HR>")

        Loop

        conn.Close() 'Fechando Conexão
        dr.Close()
        conn.Dispose()

    End Sub
```


End Class

Observando o código da **Listagem 1** localize o comando Imports System.Data e Imports System.Data.SqlClient, este comando está carregando as classes de acesso a dados, caso você fosse utilizar OleDb deveria utilizar o referido NameSpace conforme comentado no código. Efetue o Build e teste, terá o mesmo resultado da página feita em asp, porém uma página mais rápida, pois está utilizando código compilado e de acesso nativo via SqlConnection.

Digamos que você queria fazer uma rotina para excluir um registro ou grupo de registro no banco de dados, conforme já falamos nesse artigo, basta você modificar a consulta SQL colocando um comando Delete vejamos um exemplo na Listagem 2. Para testar esse código adicione uma nova página e coloque um botão (ID=btnExcluir) depois clique duas vezes no mesmo e adicione o código.

Listagem 2 - Excluindo Registros

```
Imports System.Data
Imports System.Data.SqlClient
'Imports System.Data.OleDb

Public Class temp

    Inherits System.Web.UI.Page

    Private Sub Page_Load _
        (ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

    End Sub

    Private Sub btnExcluir_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles btnExcluir.Click

        Dim conn As New SqlConnection
        Dim cmd As New SqlCommand

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "Delete [Order Details] where OrderID=10248"
        cmd.CommandType = CommandType.Text
        conn.Open()

        Dim intRetorno As Int32
```

```
intRetorno = Convert.ToInt32(cmd.ExecuteNonQuery)

If intRetorno > 0 Then
    Response.Write("Registro(s) Excluido(s):" + intRetorno.ToString)
else
    Response.write ("Resgistro não excluido")
End If

conn.Close()
conn.Dispose()

End Sub
```

```
End Class
```

Observando o código da **Listagem 2** você deve está sentindo falta do `DataReader`, como estamos acessando o banco de dados para processar uma consulta SQL que não retorna nada o mesmo não é necessário nesse caso. O próprio `Command` já retorna um valor inteiro indicando quantas linhas foram afetadas pelo comando. Repita o mesmo processo da **Listagem 2** para os outros Transacion-SQL (`Insert` e `Update`) e terá o resultado esperado. Agora fique atento à modelagem do banco, pois o mesmo vai impor e obrigar que tenha integridade nos campos, você não vai excluir um registro em uma tabela se ele está relacionado com outro em outra. Ou seja, primeiro você deve respeitar as regras do SGDB.

Agora vamos fazer um exemplo do `DataBind` que como já falamos é a capacidade do controle de ler automaticamente o retorno do banco de dados. Para isso insira uma nova página chamada de **DataGridTeste.aspx**, depois arraste um componente do tipo `DataGrid` (ID=dgConsulta) para a página conforme **Figura 08** e clique no mesmo com botão direito depois escolha `AutoFormat` para definir um layout dentre os padrões já oferecidos e adicione o código da **Listagem 3**.

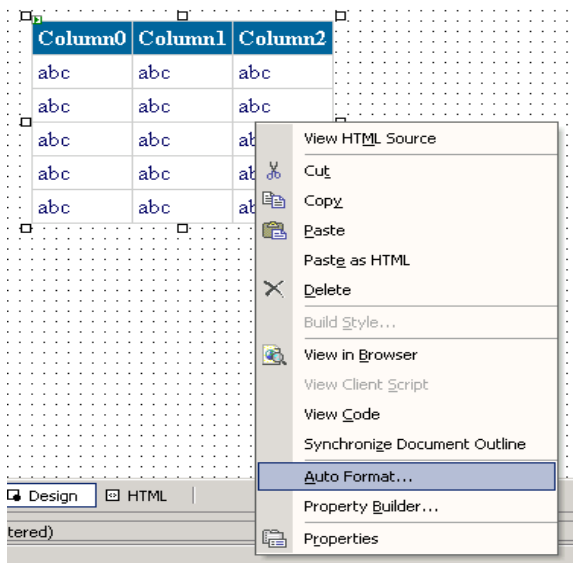


Figura 08 - Inserido DataGrid e formatando Layout

Listagem 3 - Fazendo DataBind no DataGrid

```
Imports System.Data.SqlClient
Imports System.Data

Public Class DataGridTeste

    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

        If Not Page.IsPostBack Then
            carregaDG()
        End If

    End Sub

    Sub carregaDG()
        Dim conn As New SqlConnection
        Dim cmd As New SqlCommand
        Dim dr As SqlDataReader

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "Select * from Products"
        cmd.CommandType = CommandType.Text
        conn.Open()

        dr = cmd.ExecuteReader
```

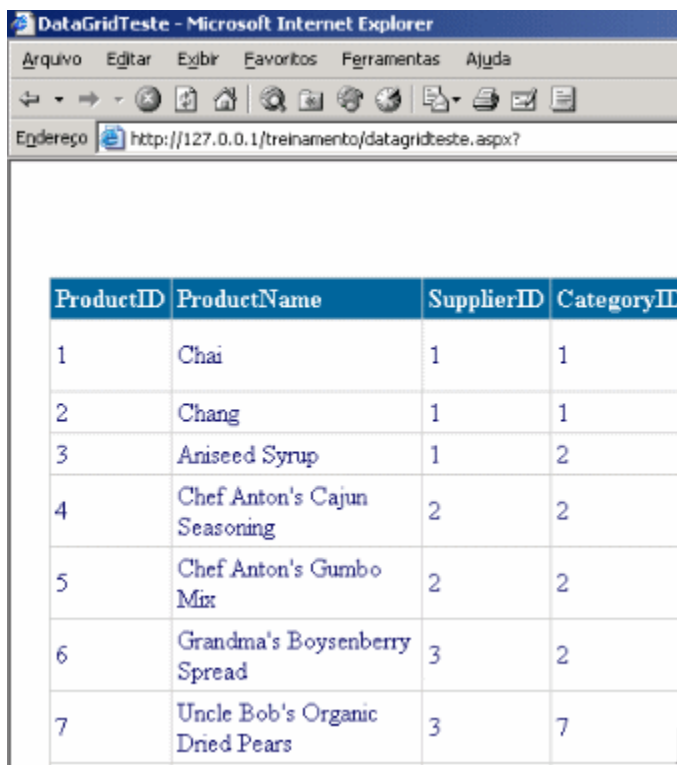
```

Me.dgConsulta.DataSource = dr
Me.dgConsulta.DataBind()
conn.Close()
dr.Close()
conn.Dispose()
End Sub

```

End Class

Observando o código da **Listagem 3** você verá apenas uma pequena novidade (Me.dgConsulta.DataSource = dr , Me.dgConsulta.DataBind()) nestas duas linhas estamos informando para o DataGrid que ele precisa ler esse DataReader e realizar o DataBind (Preenchimento automático dos dados). Efetue o build do código e verifique o resultado conforme **Figura 09**.



ProductID	ProductName	SupplierID	CategoryID
1	Chai	1	1
2	Chang	1	1
3	Aniseed Syrup	1	2
4	Chef Anton's Cajun Seasoning	2	2
5	Chef Anton's Gumbo Mix	2	2
6	Grandma's Boysenberry Spread	3	2
7	Uncle Bob's Organic Dried Pears	3	7

Figura 09 - DataGrid carregado pelo DataBind

Até o momento, todos exemplos de acesso a dados apresentados são conectados ao banco de dados. Falando ainda da classe Command já utilizamos dois métodos (ExecuteReader, Executenquery) vamos ver um outro exemplo com o método ExecuteScalar, esse método é a forma de acesso mais rápido ao banco de dados, pois independente do retorno ocasionado pela consulta SQL ele somente vai ler uma linha, uma coluna . É recomendado para retorno de Stored Procedures ou o Count(*) do Select. Confira um exemplo na **Listagem 4**.

Listagem 4 - Utilizando método ExecuteScalar da classe Command

```
Imports System.Data.SqlClient
Imports System.Data

Public Class DataGridTeste

    Inherits System.Web.UI.Page

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles Button1.Click

        Dim conn As New SqlConnection
        Dim cmd As New SqlCommand

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "Select count(*) Total from Products"
        cmd.CommandType = CommandType.Text
        conn.Open()

        Dim intRetorno As Int32

        intRetorno = Convert.ToInt32(cmd.ExecuteScalar)
        Response.Write("Total produtos:" + intRetorno.ToString)
        conn.Close()
        conn.Dispose()

    End Sub

End Class
```

O código da **Listagem 4** está utilizando o método `ExcuteScalar` para ler o retorno do `count(*)` informado pela consulta SQL.

Para tornar seu código mais claro você pode utilizar parâmetros em conjunto com as consultas SQL, essa facilidade é permitida por meio da class `SqlParameter` ou `OleDbParameter` veja o exemplo na **Listagem 5**. Nesta listagem vamos modificar a consulta sql (`Select * from Products where ProductID = 1`) para aceitar parâmetros. Como o exemplo é similar a qualquer outro estarei apenas demonstrando no código.

Listagem 05 - Utilizando parâmetros em conjunto com consulta SQL

```
Imports System.Data.SqlClient
Imports System.Data

Public Class DataGridTeste

    Inherits System.Web.UI.Page
```

```

Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
    Handles MyBase.Load

    If Not Page.IsPostBack Then
        carregaDG()
    End If

End Sub

Sub carregaDG()

    Dim conn As New SqlConnection
    Dim cmd As New SqlCommand
    Dim dr As SqlDataReader
    conn.ConnectionString = _
        "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
    cmd.Connection = conn

    'cmd.CommandText = "Select * from Products where ProductID=1"
    cmd.CommandText = "Select * from Products where ProductID=@ID"
    cmd.CommandType = CommandType.Text
    cmd.Parameters.Clear()
    cmd.Parameters.Add(New SqlParameter("@ID", SqlDbType.Int)).Value = 1
    conn.Open()

    dr = cmd.ExecuteReader
    Me.dgConsulta.DataSource = dr
    Me.dgConsulta.DataBind()

    conn.Close()

    dr.Close()

    conn.Dispose()

End Sub

End Class

```

Observando o código da **Listagem 5** veja que o código do produto que estava sendo informado na consulta passou a ser informado externamente. Em consulta SQL essa modificação não é requerida, porém para Stored Procedures (Procedimentos armazenados) é fundamental, daí uma chance de ir treinando.

Nosso próximo objetivo será executar uma Stored Procedure (procedimento armazenado). É um Transact-SQL que já fica salvo no servidor de banco de dados, com isso o banco consegue se preparar pra executar mais rapidamente esse código, por isso é muito recomendado a criação desses procedimentos dentro do SGDB. O SQL Server já

vem com um procedimento padrão chamado de sp_who2 que retorna todos usuários conectados no SGDB é muito importante para gerenciar que aplicação está utilizando SGDB no momento. Essa procedure não requer parâmetros. Veja o exemplo na **Listagem 6** que está carregando um DataGrid com retorno da procedure.

Listagem 06 - Utilizando Store Proceure sp_who2

```
Imports System.Data.SqlClient
Imports System.Data

Public Class DataGridTeste

    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

        If Not Page.IsPostBack Then
            carregaDG()
        End If

    End Sub

    Sub carregaDG()

        Dim conn As New SqlConnection
        Dim cmd As New SqlCommand
        Dim dr As SqlDataReader

        conn.ConnectionString = "Data Source=localhost; User ID=sa; Password=;
Initial Catalog=Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "sp_who2"
        cmd.CommandType = CommandType.StoredProcedure

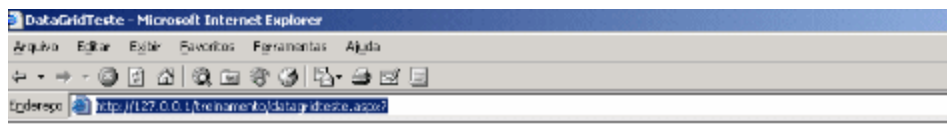
        conn.Open()
        dr = cmd.ExecuteReader
        Me.dgConsulta.DataSource = dr
        Me.dgConsulta.DataBind()

        conn.Close()
        dr.Close()
        conn.Dispose()

    End Sub

End Class
```

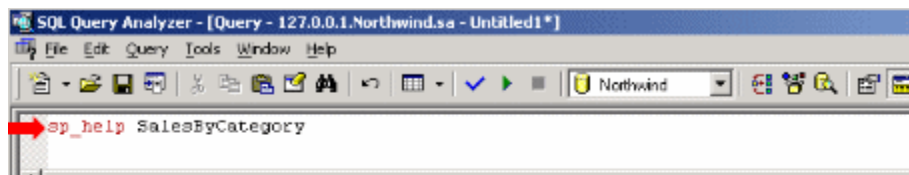
Como neste momento você já conhece muito bem o código da **Listagem 6**, a única diferença que vai encontrar é nas linhas (cmd.CommandText = "sp_who2" e cmd.CommandType = CommandType.StoredProcedure) que informa a StoreProcedure sp_who2, confira o retorno da mesma na Figura 10.



SPID	Status	Login	HostName	BlkBy	DBName	Command	CPUTime	DiskIO	LastBatch
1	BACKGROUND	sa	.	.	.	LAZY WRITER	40	0	04/11 16:17:44
2	sleeping	sa	.	.	.	LOG WRITER	10	0	04/11 16:17:44
3	BACKGROUND	sa	.	.	master	SIGNAL HANDLER	20	0	04/11 16:17:44

Figura 10 - Retorno da Stored Procedure sp_who2

Como você ficou curioso para ver como se utiliza uma Stored Procedure com parâmetros vamos ao código da **Listagem 07**. Nesse código vamos utilizar a Stored Procedure SalesByCategory presente no DataBase NorthWind, portanto não precisamos nos preocupar com sua criação até porque não é o propósito desse artigo. Essa procedure recebe dois parâmetros de entrada para você saber que parâmetros a procedure necessita utilize o comando sp_help nome_da_procedure pelo utilitário Query Analyzer conforme **Figura 11**.



Name	Owner	Type	Created_datetime
1 SalesByCategory	dbo	stored procedure	2000-08-06 01:34:53.200

Parameter_name	Type	Length	Prec	Scale	Param_order	Collation
1 @CategoryName	nvarchar	30	15	NULL	1	Latin1_General_CI_AS
2 @OrdYear	nvarchar	8	4	NULL	2	Latin1_General_CI_AS

Figura 11 - Obtendo informações da procedure SalesByCategory

Listagem 07 - Utilizando Stored Procedure SalesByCategory com parâmetro.

```
Imports System.Data.SqlClient
Imports System.Data
```

```
Public Class DataGridTeste
```

```
Inherits System.Web.UI.Page
```



```

Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
    Handles MyBase.Load

    If Not Page.IsPostBack Then
        carregaDG()
    End If

End Sub

Sub carregaDG()

    Dim conn As New SqlConnection
    Dim cmd As New SqlCommand
    Dim dr As SqlDataReader

    conn.ConnectionString = _
        "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
    cmd.Connection = conn

    cmd.CommandText = "SalesByCategory"
    cmd.CommandType = CommandType.StoredProcedure
    cmd.Parameters.Clear()
    cmd.Parameters.Add(New SqlParameter("@CategoryName",
SqlDbType.VarChar)).Value = "Beverages"
    cmd.Parameters.Add(New SqlParameter("@OrdYear", SqlDbType.VarChar)).Value
= "200"

    conn.Open()

    dr = cmd.ExecuteReader

    Me.dgConsulta.DataSource = dr

    Me.dgConsulta.DataBind()

    conn.Close()

    dr.Close()

    conn.Dispose()

End Sub

End Class

```

Como você acabou de ver na **Listagem 7** o código é parecido com o feito na **Listagem 5** pode ser conferido na **Figura 12**.

ProductName	TotalPurchase
Chai	6296,00
Chang	6299,00
Chartreuse verte	4261,00
Côte de Blaye	67324,00
Guaraná Fantástica	2318,00
Ipoh Coffee	7526,00
Lakkalikööri	6335,00
Laughing Lumberjack Lager	1445,00
Outback Lager	3395,00
Rhönbräu Klosterbier	2954,00
Sasquatch Ale	3241,00
Steeleye Stout	4632,00

Figura 12 - Retorno da StordeProcedure (SalesByCategory)

Conforme visto durante todos os exemplos de DataReader sua implementação é muito simples, basta mesmo seguir os passos apontados nesse artigo e usufruir de todas as vantagens do ADO.NET que já traz dentro de duas classes toda comunicação das APIs de cada SGBD. Deixando para o programador, simples comandos com ótimos resultados.

Utilizando o DataSet

O DataSet é uma das grandes novidades oferecidas pelo novo ADO.NET. Ele é uma estrutura de dados totalmente desconectada do banco de dados, basea-se em xml e armazenada na memória todos os dados recebidos da base de dados. Garantindo com isso toda uma manipulação fora do banco de dados para depois devolver os dados modificados. O DataSet se assemelha muito com um banco de dados pois ele armazena as informações em uma estrutura semelhante, sendo que dentro você pode criar varias tabelas, e dentro das tabelas você inclui as colunas e os dados. Ainda dentro do mesmo você pode criar relacionamentos entre as tabelas, utilizar tabelas de origem de dados diferentes, **percorrer qualquer um dos registros a qualquer momento** e inclusive transportar essa estrutura de um ponto para outro utilizando web services (Componentes web baseados em xml e soap). Verifique um modelo simplificado na **Figura 13** e **Figura 14**.

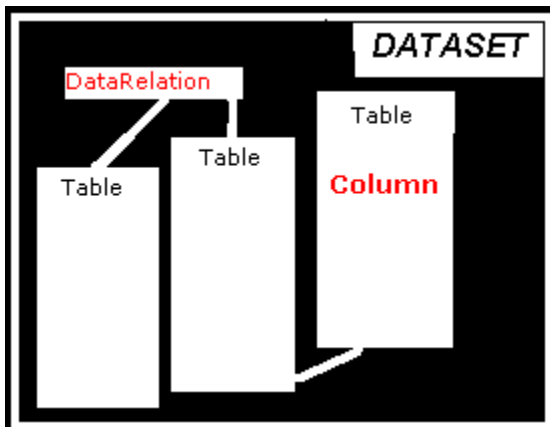


Figura 13 - Visão simplificada Data Set

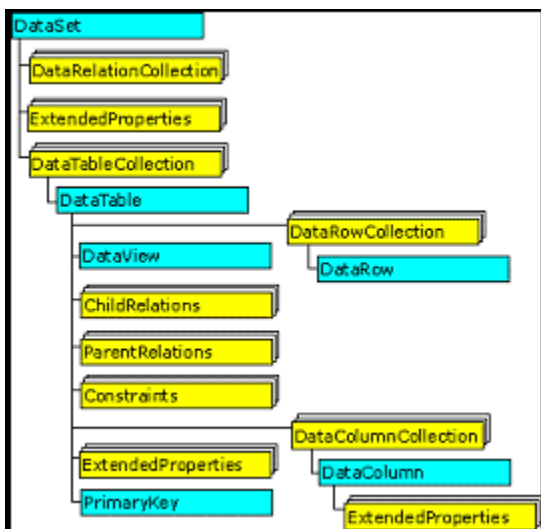


Figura 14 - Visão simplificada do Data Set

Para utilizarmos o Data Set vamos precisar seguir os passos que fizemos para o DataReader com algumas modificações conforme os tópicos abaixo:

1. **Connection** - Abre e fecha conexão, necessita de uma string de acesso conforme Tabela 04.
2. **Command** - Vai processar seu comando Transaction-SQL no SGDB.
3. **DataAdapter** - Obtém dados do SGDB e preenche o DataSet
4. **DataSet** - Estrutura de dados xml, será preenchida pelo DataAdapter

Como você já utilizou o DataReader o DataSet também é de simples utilização, vamos fazer o mesmo exemplo utilizando uma página feita em asp e convertendo para asp.net utilizando ADO.NET e DataSet.

```
<%
'Arquivo: Teste.asp
set conn = server.createobject("adodb.connection")
Conn.Open "Provider=SQLOLEDB.1;Data Source=localhost;Initial
Catalog=Northwind;User ID=sa;Password=;"
```

```

sql="select * from Products"
set rs=Conn.Execute(SQL)
do while not rs.eof
response.write cstr(rs.fields("ProductID"))+"<BR>"
response.write cstr(rs.fields("ProductName"))+"<hr>"
rs.movenext
loop
Conn.Close
rs.Close
set rs=nothing
set conn=nothing
%>

```

Agora adicione uma nova página webform ao seu projeto com o nome de testedataset.aspx, depois efetue dois cliques e adicione o código da **Listagem 8**.

Listagem 8 - Código para testedataset.aspx

```

Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.OleDb

Public Class testedataset

    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        Dim conn As New SqlConnection 'Criando objeto Connection
        Dim cmd As New SqlCommand 'Criando objeto Command
        Dim da As New SqlDataAdapter 'Criando DataAdapter
        Dim ds As New DataSet 'Criando objeto DataSet

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "Select * from Products"
        cmd.CommandType = CommandType.Text
        da.SelectCommand = cmd
        da.Fill(ds, "Produtos") 'Preenchendo DataSet

        'Desconectado do banco de dados
        conn.Dispose()
        da.Dispose()

        'Lendo DataSet
        'Neste ponto não estamos mais conectados ao banco de dados

        Dim intl As Int32

```

```
For intl = 0 To ds.Tables(0).Rows.Count - 1
    Response.Write(Convert.ToString(ds.Tables(0).Rows(intl)(0)) + "<BR>")
    Response.Write(Convert.ToString(ds.Tables(0).Rows(intl)(1)) + "<HR>")
Next
```

```
End Sub
```

```
End Class
```

Observando o código acima vamos encontrar muita semelhança com o primeiro exemplo de DataReader apresentado na **Listagem 01**, a principal diferença que você vai encontrar é que não estamos chamando o método Conn.Open pois isso está sendo feito pelo DataAdapter que se encarrega de acessar os dados e preencher o DataSet para que possa ser trabalhando de forma desconectada. Porém se você explicitamente utilizar o Conn.Open terá que efetuar o Conn.Close. Logo em seguida no final no loop que estamos realizando colocamos para varrer até a ultima linha do DataSet, de forma que obtemos essa quantidade pelo Rown.Count. Neste exemplo estamos fazendo referências a todos itens do DataSet utilizando "Indices" no lugar de colocar o nome das colunas e tabelas.

Conforme já afirmamos anteriormente é possível criar um DataSet utilizando a IDE do Visual Studio para que de forma visual possamos estabelecer o acesso ao SGDB. Para montarmos esse exemplo siga os passos abaixo:

1. Adicione um novo webform chamado de testeide.aspx
2. Acione o Server Explorer conforme Figura 15. Utilizando o menu view>server explorer ou CTRL+ALT+S

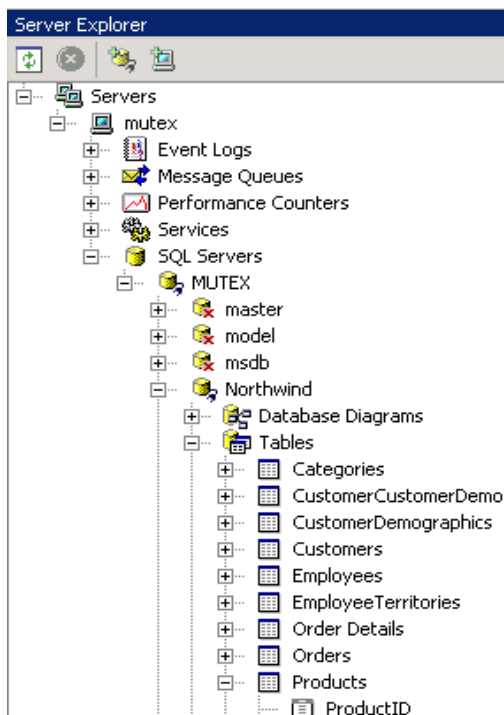


Figura 15 - Server Explorer

O Server Explorer é um novo utilitário oferecido pela IDE do Visual Studio que permite a gerência do banco de dados, ele funciona praticamente com todos os banco de dados compatíveis com OLEDB. É uma ferramenta muito útil, pois no próprio Visual Studio você tem recursos oferecidos pelo Enterprise Manager (**Figura 2**) do SQL Server. Você pode criar tabelas, colunas, modificar e exibir dados conforme **Figura 16**.

Após esse breve resumo sobre o Server Explorer, clique na tabela products e arraste diretamente para o formulário, automaticamente será criado um SqlConnection e um SqlDataAdapter conforme **Figura 17**.

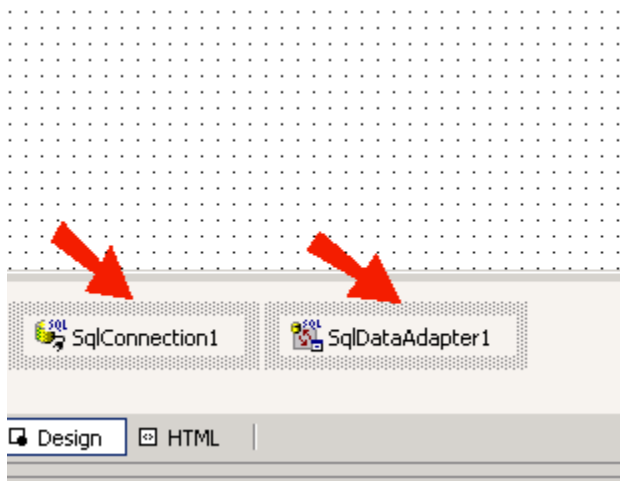


Figura 17 - SqlConnection e SqlDataAdapter criados pelo IDE

O próximo passo agora é criar o DataSet, para isso basta clicar no botão direito do SqlDataAdapter e escolher a opção "Generate DataSet" conforme **Figura 18** e **Figura 19**.

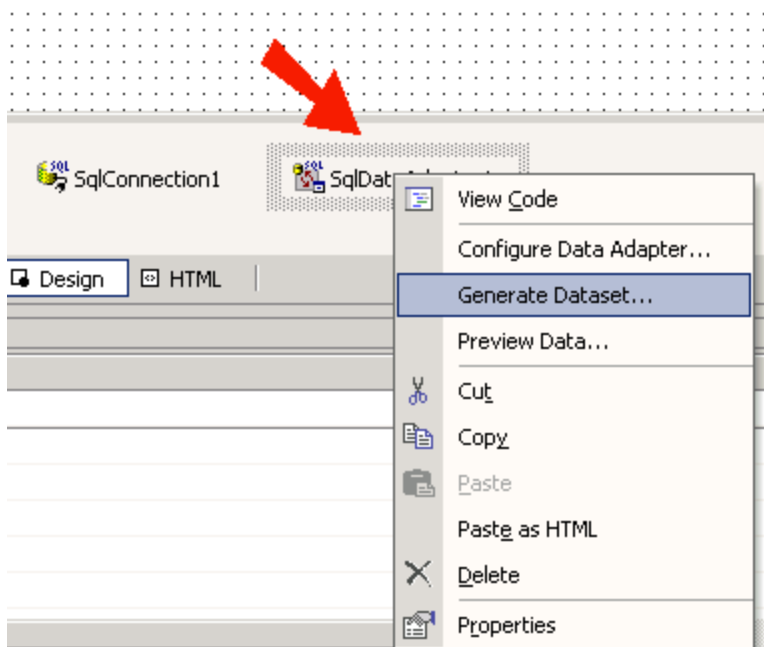


Figura 18 - Criando DataSet pelo IDE

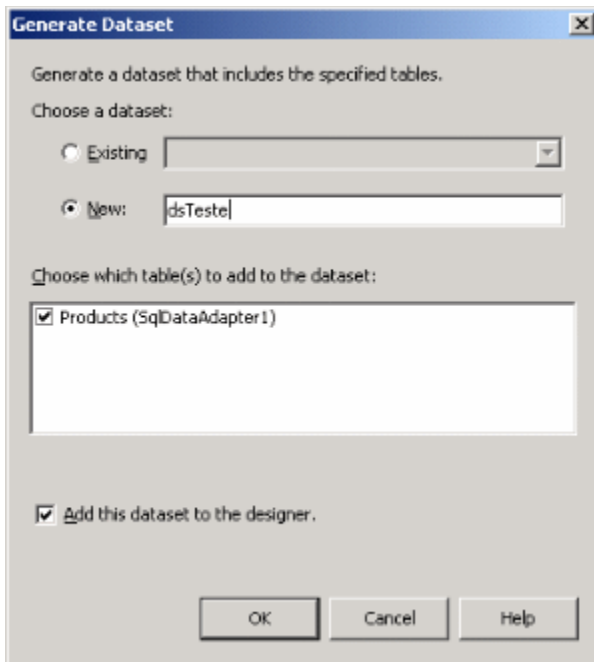


Figura 19 - Criando DataSet pelo IDE

Concluindo esse passo você já criou o DataSet e ele já deve estar aparecendo na IDE do Visual Studio. A grande novidade do .NET é que todo o código para criar todos os passos que seguimos até o momento está visível, basta efetuar dois cliques no formulário e ir a área de código depois clique na Região "Web Form Designer Generated Code", com isso você vai acompanhar todos os passos que o Visual Studio está fazendo até criar seu DataSet. Todo procedimento que estamos fazendo aqui é similar ao da **Listagem 8** com a diferença que estamos utilizando a interface visual para criar. Veja agora a mesma representação na **Listagem 9** que difere da **Listagem 8** por não precisar criar os objetos de acesso a dados sendo que já foi criado pelo IDE.

Listagem 9 - Utilizando IDE do Visual Studio para gerar DataSet

```
Public Class testeide
    Inherits System.Web.UI.Page
    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

        'Opcionalmente estou trocando a String de conexão porque
        'o padrão do IDE é Windows Integrado, e a máquina precisa estar
        'corretamente configurada.

        Me.SqlConnection1.ConnectionString = _
            "Data Source=localhost;user id=sa;Password=;Initial catalog=Northwind;"

        Me.SqlDataAdapter1.Fill(Me.DsTeste1)

        Dim intl As Int32
        For intl = 0 To DsTeste1.Tables(0).Rows.Count - 1
```

```

Response.Write(Convert.ToString(DsTeste1.Tables(0).Rows(intI)(0)) + "<BR>")
Response.Write(Convert.ToString(DsTeste1.Tables(0).Rows(intI)(1)) + "<HR>")
Next

```

```

End Sub
End Class

```

Observado as diferenças entre os códigos, comente o código na página (testeide.aspx) e arraste para página um componente DataGrid (ID=dgConsulta) alterando seu ID conforme especificado. Depois altere as propriedades DataSource e DataMember conforme **Figura 20**. Observe que está mudando na janela de propriedades, isso está sendo possível porque você criou o DataSet pelo IDE. Depois mude a aparência do DataGrid clicando com botão direito em cima do mesmo e escolhendo AutoFormat.

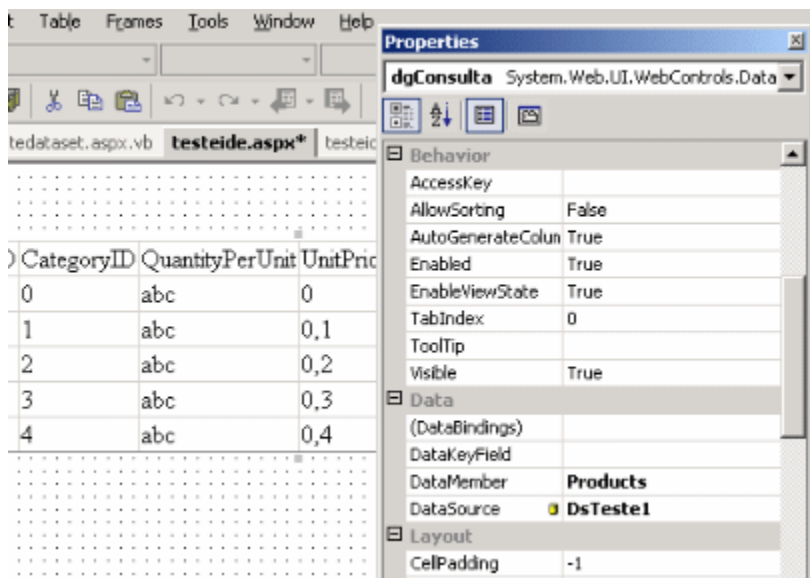


Figura 20 - Configurando DataGrid

Configurado o DataGrid, vamos ao código na Listagem 10 para ativar o funcionamento.

Listagem 10 - Carregando DataGrid

```

Public Class testeide
    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

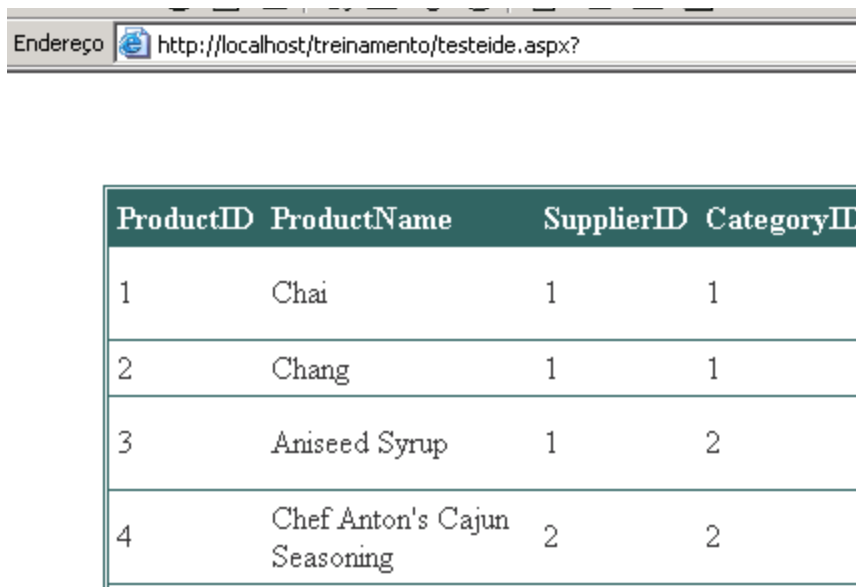
        Me.SqlConnection1.ConnectionString = _
            "Data Source=localhost;user id=sa;Password=;Initial Catalog=Northwind;"
        Me.SqlDataAdapter1.Fill(Me.DsTeste1)
        Me.dgConsulta.DataBind()

    End Sub

```


End Class

Após adicionar o código efetue o Build e teste vai notar que apenas com essas linhas de código você vai carregar os dados no DataGrid conforme **Figura 21**.



The screenshot shows a web browser window with the address bar displaying "http://localhost/treinamento/testeide.aspx?". Below the browser, a DataGrid is displayed with the following data:

ProductID	ProductName	SupplierID	CategoryID
1	Chai	1	1
2	Chang	1	1
3	Aniseed Syrup	1	2
4	Chef Anton's Cajun Seasoning	2	2

Figura 21 - DataGrid usando IDE

A vantagem de se utilizar o IDE é poder realizar as configurações de acesso ao banco de dados pela janela de propriedades e reduzir a quantidade de código a ser digitado. Não implica na redução de código gerado, pois o Visual Studio gera uma quantidade enorme de código para permitir essas facilidades.

No entanto nunca utilizo esse recurso da IDE, pois considero muito fácil o acesso utilizando o código direto e fica mais claro o entendimento. E se você for seguir o padrão patterns & practices da Microsoft, toda implementação do Data Access Block é feita utilizando código. Fica a sua escolha, escolher o caminho.

Resumindo o que já fizemos até esse momento, nós carregamos um DataSet com os dados de um SGDB. Mas sabemos que o DataSet contém DataTable que contém DataColumn e DataRow conforme já visto na **Figura 13** e na **Figura 14**. Vamos agora criar um DataTable totalmente via código e inserir dados via código. Para isso insira um novo webform (datatablecodigo.aspx) e depois adicione o código da **Listagem 11**.

Listagem 11 - Criando DataTable usando código

```
Public Class DataTableCodigo
    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
        System.EventArgs) _
        Handles MyBase.Load

        If Not Me.Page.IsPostBack Then
```

```

        carrega()
    End If

End Sub

Sub carrega()

    Dim dt As New DataTable

    'Criando Coluna Coódigo
    'com auto numeração.

    Dim dc As New DataColumn
    dc = New DataColumn
    dc.DataType = System.Type.GetType("System.Int32")
    dc.ColumnName = "Codigo"
    dc.ReadOnly = True 'Não pode ser alterado
    dc.Unique = True 'Não repete
    dc.AutoIncrement = True ' Define AutoIncremento
    dc.AllowDBNull = False 'Não aceitar nulos
    dt.Columns.Add(dc) 'Adicionando Coluna

    'Criando Coluna Nome
    dt.Columns.Add(New DataColumn("Nome", GetType(String)))
    'Criando Coluna Email
    dt.Columns.Add(New DataColumn("Email", GetType(String)))

    '----- Fim Criação DataTable -----

    'Adicionando Linhas

    Dim dr As DataRow = dt.NewRow
    dr("Nome") = "Ramon Durães"
    dr("Email") = "ramonduraes@mutex.com.br"
    dt.Rows.Add(dr)
    dr = dt.NewRow
    dr(1) = "Mike Silver"
    dr(2) = "mk@msspace.com"
    dt.Rows.Add(dr)

    '////////////////////
    'Criando DataGrid Dinamicamente
    'e adicionando a página
    '////////////////////

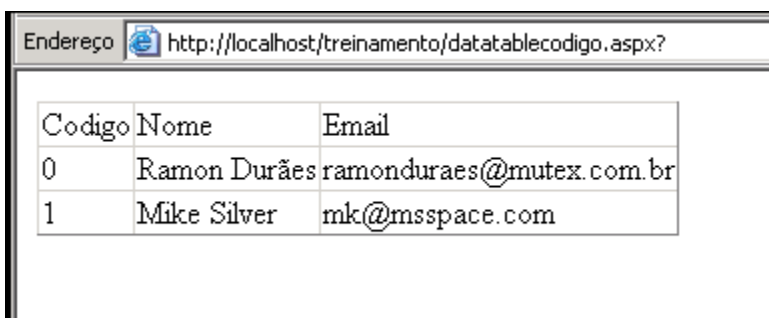
    Dim dg As New DataGrid
    dg.DataSource = dt.DefaultView
    dg.DataBind()
    Me.Page.FindControl("Form1").Controls.Add(dg)

End Sub

```

End Class

Após adicionar o código efetue o Build e teste essa página, o resultado deve ser similar a **Figura 22**. Neste exemplo estamos criando um DataTable, adicionando as colunas, e depois as linhas. Tudo isso feito sem nenhum acesso ao SGDB. O detalhe importante é a coluna "Codigo" que foi criada como auto-numeração e o DataTable automaticamente adiciona o numero a coluna. Observe que nesse exemplo não utilizamos DataSet.



Codigo	Nome	Email
0	Ramon Durães	ramonduraes@mutex.com.br
1	Mike Silver	mk@msspace.com

Figura 22 - DataTable Via código

Digamos agora que você queira ver o conteúdo do DataTable que está armazenado na memória, então vamos salvar o conteúdo do mesmo em um arquivo XML conforme **Listagem 12**. Essa Listagem é semelhante a anterior e poderia ser qualquer outra que se tenha um DataSet pois o mesmo que esta no formato de XML, para esse caso criamos um DataSet e armazenamos o DataTable dentro. O Xml gerado é mostrado na **Figura 23**.

Listagem 12 - Gerando arquivo xml apartir de um DataSet

```
Public Class DataTableCodigo
    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

        If Not Me.Page.IsPostBack Then
            carrega()
        End If

    End Sub

    Sub carrega()

        Dim dt As New DataTable

        'Criando Coluna Coódigo
        'com auto numeração.

        Dim dc As New DataColumn
        dc = New DataColumn
```

```

dc.DataType = System.Type.GetType("System.Int32")
dc.ColumnName = "Codigo"
dc.ReadOnly = True 'Não pode ser alterado
dc.Unique = True 'Não repete
dc.AutoIncrement = True ' Define AutoIncremento
dc.AllowDBNull = False 'Não aceitar nulos
dt.Columns.Add(dc) 'Adicionando Coluna

'Criando Coluna Nome
dt.Columns.Add(New DataColumn("Nome", GetType(String)))
'Criando Coluna Email
dt.Columns.Add(New DataColumn("Email", GetType(String)))

'----- Fim Criação DataTable -----

'Adicionando Linhas

Dim dr As DataRow = dt.NewRow
dr("Nome") = "Ramon Durães"
dr("Email") = "ramonduraes@mutex.com.br"
dt.Rows.Add(dr)
dr = dt.NewRow
dr(1) = "Mike Silver"
dr(2) = "mk@msspace.com"
dt.Rows.Add(dr)

'////////////////////
'Criando DataGrid Dinamicamente
'e adicionando a página
'////////////////////

Dim dg As New DataGrid
dg.DataSource = dt.DefaultView
dg.DataBind()
Me.Page.FindControl("Form1").Controls.Add(dg)

'////////////////////
'Criando XML
'////////////////////

Dim ds As New DataSet
ds.Tables.add(DT)
ds.WriteXml("C:\temp\datasetcodigo.xml")

End Sub

End Class

```

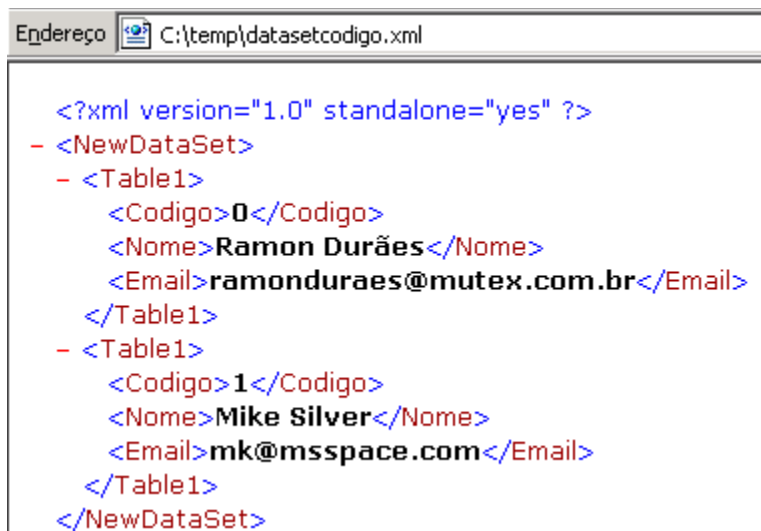


Figura 23 - XML Gerado pelo WriteXML

O DataSet possui dois métodos importantes o WriteXML e o ReadXml, sendo usados para escrita e para leitura, por está no formato xml, DataSet pode ser transportado pela internet por meio de web services.

Vamos agora fazer o contrário, vamos carregar um DataSet a partir de um arquivo xml utilizando método ReadXML, para isso vamos criar um novo XML, então botão direito no projeto (solution explorer)>Add new item>XML conforme **Figura 24**, e dê o nome de demoxml.xml.

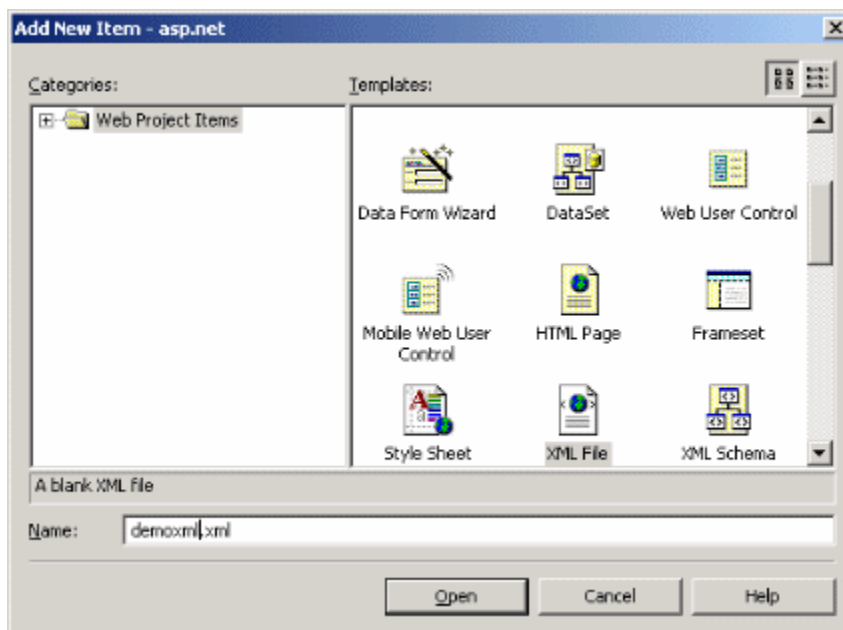


Figura 24 - Criando arquivo xml

Depois digite o código xml conforme a **Figura 25**. Após adicionar essa parte, clique na opção Data logo no final e terá o resultado mostrado na **Figura 26**.

Data:

Data for Usuarios		
	Codigo	Nome
▶	1	Ramon Durãe
*		

Figura 26 - Adicionando registros

Você pode adicionar agora mais registros usando essa interface visual, após terminar clique novamente em xml e vai ver o código gerado. Após terminar insira um novo webform (datasetread.aspx) e adicione o código da **Listagem 13**.

Listagem 13 - Carregando DataSet usando xml


```
Public Class datasetread
    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        Dim ds As New DataSet
        ds.ReadXml(Server.MapPath("demoxml.xml"))
        dgConsulta.DataSource = ds.Tables(0)
        dgConsulta.DataBind()
    End Sub

End Class
```

Após efetuar Build e testar o código teremos o resultado na **Figura 27**.

Endereço  http://localhost/treinamento/datasetread.aspx?

Codigo	Nome
1	Ramon Durães

Figura 27 - Lendo xml e carregando DataSet

Conforme visto com poucas linhas já temos um DataSet que está sendo alimentado a partir de um arquivo XML. Vamos agora criar um DataGrid com paginação e utilizando um DataSet. Para iniciar adicione um novo WebForm (DataGridpagina.aspx), depois adicione um DataGrid (ID=dgConsulta) e formate com layout preferido, depois clique no mesmo com botão direito e escolha Property Builder>Paging e configure conforme a **Figura 28**. Feito isso efetue dois cliques na página e adicione o código de acordo com a **Listagem 14**.

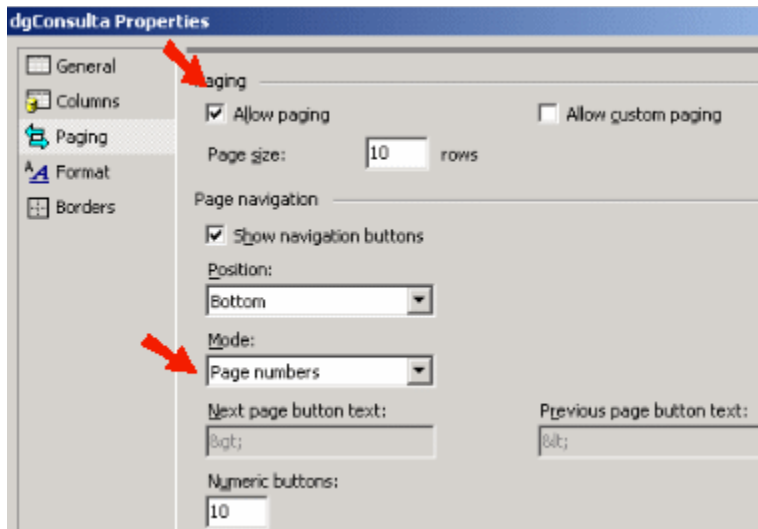


Figura 28 - Configurando DataGrid para paginação

Listagem 14 - paginando DataGrid com DataSet

```
Imports System.Data
Imports System.Data.SqlClient
Public Class datagridpagina

    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

        If Not Page.IsPostBack Then
            carregaDG()
        End If

    End Sub

    Sub carregaDG()

        Dim conn As New SqlConnection
        Dim cmd As New SqlCommand
        Dim da As New SqlDataAdapter
        Dim ds As New DataSet

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "Select * from Products"
        da.SelectCommand = cmd
        da.Fill(ds)

        conn.Dispose()
```

```

cmd.Dispose()
da.Dispose()

Me.dgConsulta.DataSource = ds
Me.dgConsulta.DataBind()

End Sub

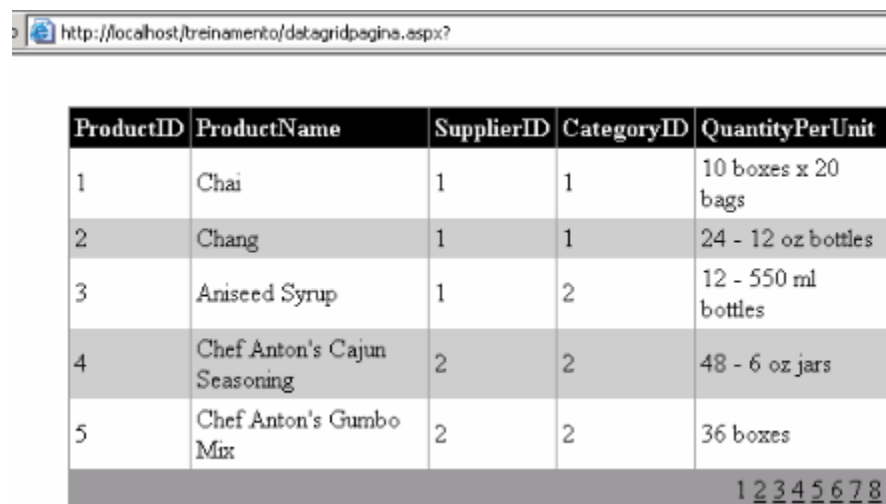
Private Sub dgConsulta_PageIndexChanged(ByVal source As Object, _
    ByVal e As System.Web.UI.WebControls.DataGridPageChangedEventArgs)
    '
    ' Handles dgConsulta.PageIndexChanged

    Me.dgConsulta.CurrentPageIndex = e.NewPageIndex
    carregaDG()
End Sub

End Class

```

Após adicionar o código e efetuar Build confira o resultado semelhante a **Figura 29**. Fique apenas atento ao evento `PageIndexChanged` do `DataGrid` pois é mesmo que é disparado quando se clica na mudança de página no `DataGrid`.



http://localhost/treinamento/datagridpagina.aspx?

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit
1	Chai	1	1	10 boxes x 20 bags
2	Chang	1	1	24 - 12 oz bottles
3	Aniseed Syrup	1	2	12 - 550 ml bottles
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars
5	Chef Anton's Gumbo Mix	2	2	36 boxes

1 2 3 4 5 6 7 8 9

Figura 29 - DataGrid com paginação

Falando sobre `DataSet` não podemos também de deixar de fazer um exemplo sobre `DataView` que um padrão de projeto Model-View, que permite a criação de visões dos dados recuperados, podendo filtrar, ordenar os dados que estão no `DataTable/DataSet`. Existe alguns casos que você não consegue mudar a ordem dos dados, como o retorno de uma procedure. Então você usa o `DataView` para mudar a ordem dos dados. Vamos agora aproveitar e fazer um exemplo que englobe duas coisas `DataView` e `StoredProcedures`. Para essa tarefa adiciona um novo webform (`DataSetView.aspx`), depois adicione um `DataGrid` (`ID=dgConsulta`) e adicione o código da **Listagem 15**.

Listagem 15 - Utilizando `DataView`


```

Imports System.Data
Imports System.Data.SqlClient
Public Class datasetview

    Inherits System.Web.UI.Page

    Private Sub Page_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) _
        Handles MyBase.Load

        carregaDG()
    End Sub

    Sub carregaDG()

        Dim conn As New SqlConnection
        Dim cmd As New SqlCommand
        Dim da As New SqlDataAdapter
        Dim ds As New DataSet

        conn.ConnectionString = _
            "Data Source=localhost; User ID=sa; Password=;Initial Catalog= Northwind;"
        cmd.Connection = conn
        cmd.CommandText = "SalesByCategory"
        cmd.CommandType = CommandType.StoredProcedure
        cmd.Parameters.Clear()
        cmd.Parameters.Add(New SqlParameter("@CategoryName",
SqlDbType.VarChar)).Value = "Beverages"
        cmd.Parameters.Add(New SqlParameter("@OrdYear", SqlDbType.VarChar)).Value
= "2002"

        da.SelectCommand = cmd
        da.Fill(ds)
        conn.Dispose()
        da.Dispose()
        cmd.Dispose()

        'Utilizando DataView para ordenar
        Dim dv As New DataView(ds.Tables(0))
        dv.Sort = "ProductName DESC"
        Me.DataGrid1.DataSource = dv
        Me.DataGrid1.DataBind()

    End Sub

End Class

```

Após adicionar o código e efetuar o Build confira na **Figura 30**, neste exemplo ordenamos pela coluna ProductName, isso dentro do DataView e não no banco de dados, até porque

o retorno do mesmo é uma procedure. E estamos passando o DataView para o DataGrid e não o DataSet.



The screenshot shows a web browser window with the address bar displaying 'http://localhost/treinamento/datasetview.aspx?'. The main content area contains a table with two columns: 'ProductName' and 'TotalPurchase'. The table lists 12 products, sorted by total purchase value in descending order. A red arrow points to the 'ProductName' header.

ProductName	TotalPurchase
Steeleye Stout	4632,00
Sasquatch Ale	3241,00
Rhönbräu Klosterbier	2954,00
Outback Lager	3395,00
Laughing Lumberjack Lager	1445,00
Lakkaikööri	6335,00
Ipoh Coffee	7526,00
Guaraná Fantástica	2318,00
Côte de Blaye	67324,00
Chartreuse verte	4261,00
Chang	6299,00
Chai	6296,00

Figura 30 - Exibindo DataView

Conforme resultado na **Figura 30**, o retorno do banco de dados está ordenado.