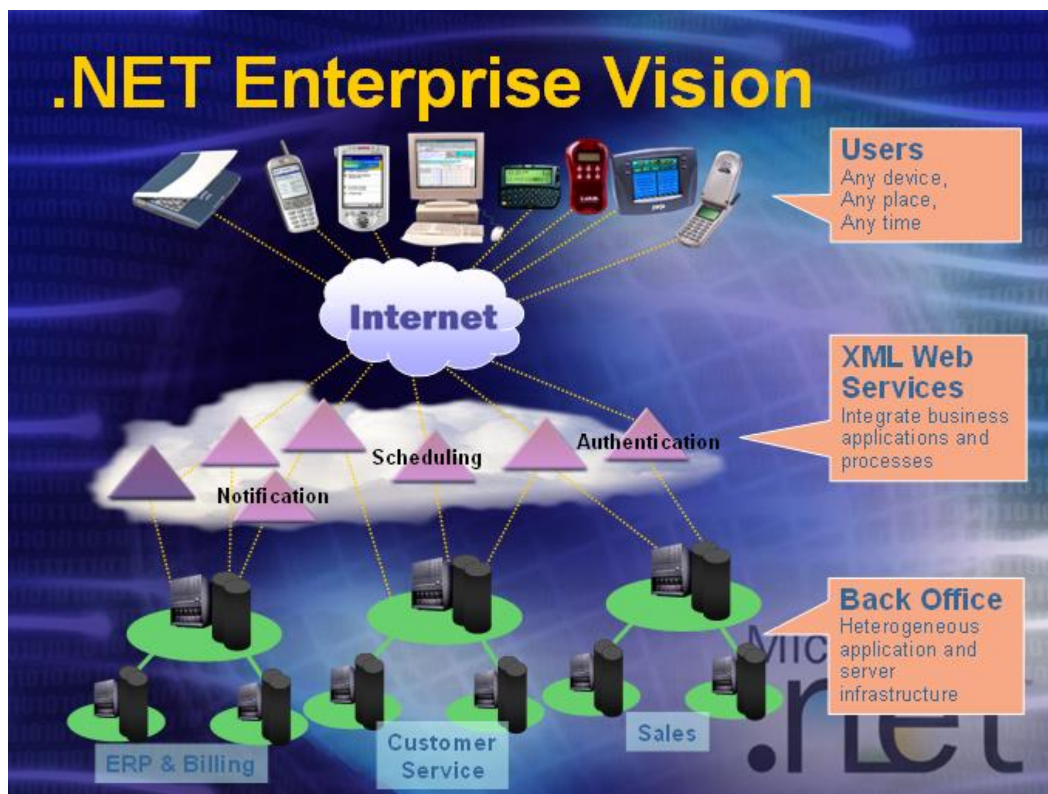
1. Overview of Microsoft .NET Programming

1.1 The .NET Framework

What is .Net Platform?

Microsoft .NET is a software development platform based on virtual machine architecture. Dot Net Platform is:

- **Language Independent** – dot net application can be developed different languages (such as C#, VB, C++, etc.)
- **Platform Independent** – dot net application can be run on any operating system which has .net framework installed.
- **Hardware Independent** – dot net application can run on any hardware configuration
  It allows us to build windows based application, web based application, web service, mobile application, etc.



How .Net Address Today's Challenges

Clearly, business users today are faced with a lot of technology, but a limited ability to get at their to get at their data in meaningful, productive way.

What are the benefits of .Net?

1. Simplify Application Development
2. Simplify Application Development
3. XML everywhere
4. Universal Data Access
5. Web Service: Collaboration over the internet

The Building Blocks of .Net are:

- The .Net Framework
- .Net Enterprise Servers
- .Net Building Block Services
- Visual Studio .Net

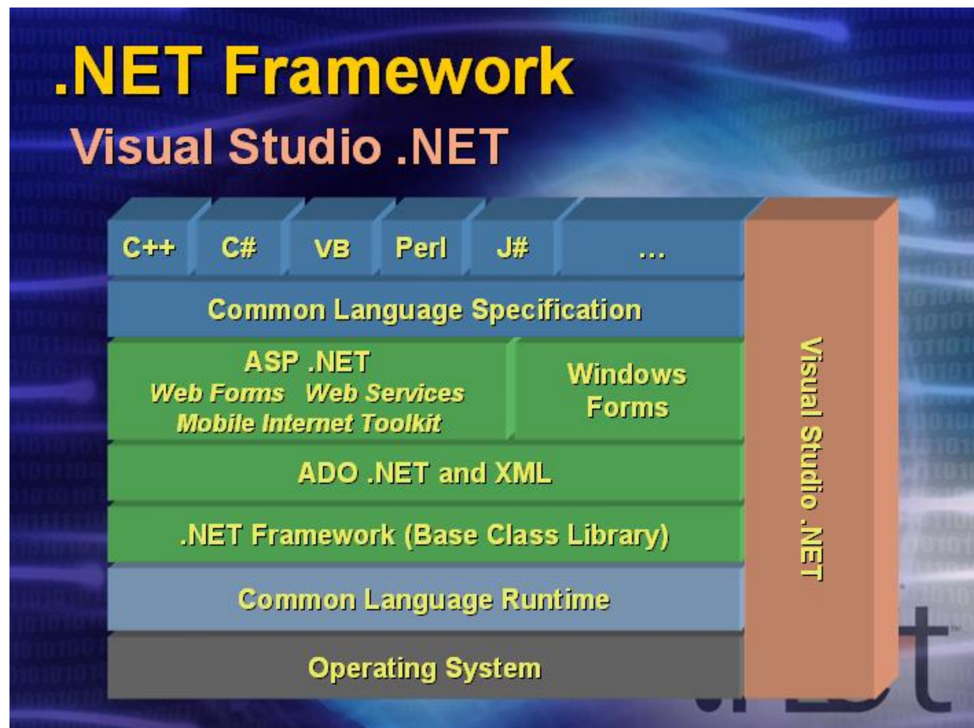
What is .Net Framework?

.Net framework provided rich set of functionality classes, Assemblies, Data types and services and also simplified application development & deployment.

OR

.Net Framework provides a foundation upon which .net application and xml web services are built and executed.

For developers, Microsoft provides the new .Net Framework, which is a set of system secure class and data type that enhance developer productivity and give easier access to the deep set up of functionality provided by windows. The .Net Framework is shown in the figure bellows:



The .Net Framework is a layered system of classes and service that starts with the operating system service, and moves up through a set of system class (the Base Class Library) and abstracted classes(For example ASP.NET).

- **Common Language Runtime:** A rich runtime environment that handles important runtime tasks for the developer, including memory management and garbage collection. Built around the common Type System and defines a common type system for all language.
- **Base Class Library:** A rich set of functional base class that may be inherited and extended by other classes in Framework. For example, System.Object provides base object functionality that all classes in the Framework inherit. System.IO provides serialization in end from different Input/Output devices, such as files and stream.
- **Extended Class Libraries:** class libraries that are focused on one aspect of development. These classes are extended from the Base Class Library, and are design to make it easier and faster to developed a specific type of application. For example, ASP.NET include classes that are focused on developing Web Services. Other example, include ADO.NET(for data access), XML.NET in braces to parse and manipulate DOCs) and Windows Forms (the successor to VB forms).
- **Common Language Specification:** define requirements for .Net Languages, by specifying a set of rules that .Net compliant Languages must follow. One of these rules in that the language must ad here to common type system.
- **Multiple Programming Language:** VB.NET, C#.NET and C++.NET are just some of the many languages that are available for coding in .Net. The .Net Framework provides one platform and unified programming model for several languages. Java is conspicuously absent from the .Net family of language, probably due to the licensing dispute between Sun Microsystems and Microsoft.
- **Visual Studio .Net:** an integrated development environment for coding with the .Net Framework. The diagram shows VS.NET spanning the entire .Net Framework because it provides tools that access each part of the Framework.
- **Windows and COM+ Service:** There are technically not part of the .NET Framework, but they are a requirement for today's .NET Framework SDK.

To summarize, the important concept behind the .NET Framework are:

- Built on a common set of Framework classes

- Provides a Common Type System, that is the cornerstone of unified programming model for all .NET compliant languages
- Provides a Common Language Runtime that provides runtime service for components and applications
- Provides extended class libraries for ASP.NET, ADO.NET, XML.NET and Windows Forms
- Visual Studio.NET is an integrated development environment for the NET Framework

The .NET Enterprise Servers:

Microsoft is orienting all of their recent and upcoming technology around .NET. To this end they have identified a suite of products called **.NET Enterprise Servers,** which are server-based application that web enable enterprise systems. These include applications that you may already be using but did not realize were part of .NET initiative. Example of .NET Enterprise Servers include:

- Windows 2000 Advance Server
- Application Center 2000/2008
- SQL Server 2000/2005/2008
- Exchange Server 2000
- Host Integration Server 2000
- Internet Security and Acceleration Server 2000/2008
- Commerce Server 2000
- BizTalk Server

.NET Building Block Services

The .NET building block services will include:

- Authentication
- Notification and Messaging
- Directory and Search
- Calendar
- XML Store

Visual Studio .NET

Visual Studio .Net is newest version of Microsoft's development toolkit for creating .NET solutions. It is designed to promote **Rapid Application**

Development (RAD). The .NET Framework SDK actually provides everything that you need. However, you will miss out on many of the benefits that Visual Studio.NET provides: an integrated development environment and tight integration with the .NET Framework.

The key features of Visual Studio.NET are:

- Full integration with the .NET Framework
- Integrated development environment
- Mixed Language development including cross language debugging
- RAD features for application development
- Visual Designers for XML, HTML and Data
- Expanded debugging across projects, including store procedures

Overview of .NET Applications

There are several types of applications that you can build with .NET:

- **Windows Forms Applications:** Windows Form Applications are the newest generation of the traditional windows-based applications that provide a form-based user interface and n-Tier, partitioned architecture. Windows Forms are object that are derived from the .NET Framework.
  Windows Forms provide the following useful feature:
  - **A new Forms Architecture:** an object oriented set of classes including the base Forms class
  - **The Control object Model:** a set of Windows Controls for the user interface
  - **A new Event Model:** A set of events based on delegate which are similar to callbacks
- **Windows Forms Controls:** Windows Form Controls are the successors to ActiveX controls. They are reusable components that provide a user interface and responsive to user events
- **Windows Service Application:** Windows Service applications were formerly known as NT services. They are executables that run in independent windows sessions with no user interaction. Microsoft developers will be most familiar with the following services:
  - Distributed Transaction Coordinator
  - IIS Admin Service
  - Simple Mail Transport Protocol (SMTP)
  - Task Scheduler
  - Windows Installer
  - World Wide Web publishing Service

- **ASP.Net Web Applications:** ASP.NET is the next generation platform for developing web application.
  ASP.NET provides the following two programming models:
  - **Web Forms:** these are analogous to Windows Forms, and even provide Web Controls that can be dropped on to the form to provide a user interface, and to automate common functions the functionality of client – side script.
  - **Web Services:** these are remote application components that receive and respond to requests using open standard protocols, RPC calls over HTTP using XML (combined into SOAP envelopes).
- **Web Services:** ASP.NET and the .NET Framework together provide classes and services for building web services components.
- **Web Simple Description Language (WSDL)** allows outside consumers to gather the information they need to communicate with your web service.
- **Simple Object Access Protocol (SOAP):** The SOAP specification defines how to send XML over HTTP. Requests and response to and from Web Services are formatted and Passed via SOAP envelops.
- **U**

1.2 The Common Language Runtime (CLR):

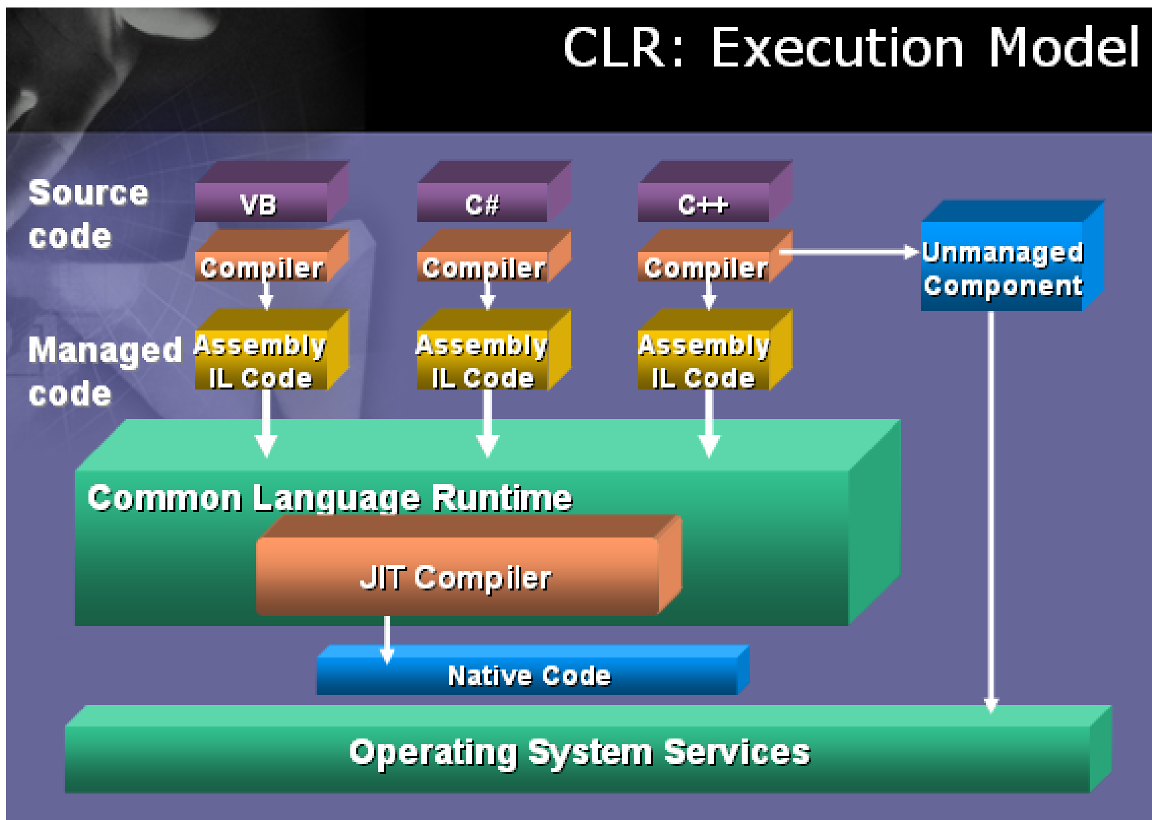
Definition:
Common Language Runtime (CLR) is execution engine of .Net Framework based application.Code that runs under the control of the CLR is called **managed code**, because the CLR defines the rules that code's development language must conform to specifically, the CLR works with two other Framework services to define the rules for .NET languages. These are:

- **Common Type System (CTS):** defines standard reference and value types that are supported in the .NET Framework
- **Common Language Specification (CLS):** defines rules that a development must comply with in order to be managed by the .NET Framework

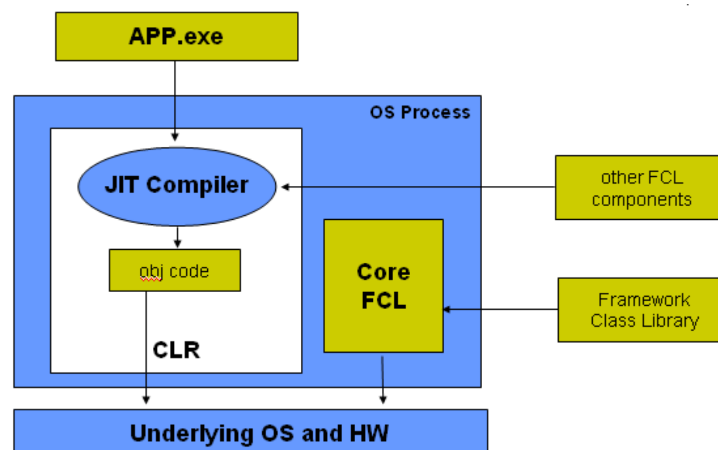
The Specific benefits of the .NET Framework are:

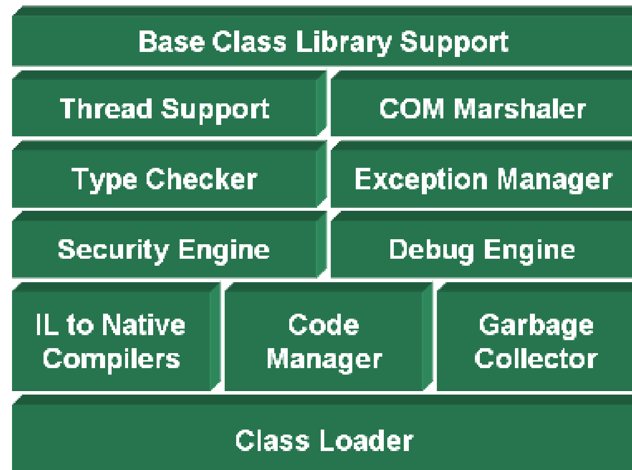
- DLL "Heaven" not "DLL Hell"
- Component integration replaces interface
- Simplified deployment
- Improved resource management
- Multiple language integration
- Unified, extensible Class Library

- Structured exception handling



The Common Language Runtime will manage code for any language that conforms to the CLS and the CTS. .NET code is compiled in a two-step process whereby the code is first converted into a language neutral generalized instruction set called Microsoft Intermediate Language (IL). Next, the Just-In-Time executed by the CLR. .NET code is compiled into assemblies, which are similar to dynamic link libraries (DLLs) except that they hold Meta data and self-describing.





The CLR provides a number of runtime support service using **Virtual Execution System (VES).** The VES is responsible for implementing and enforcing the Common Type System. The execution engine uses Meta data information to understand the structure of the components.

The specific components of the VES are:

- Class Loader (Load managed Code)
- Microsoft Intermediate Language (MSIL)
- MSIL-to-Native code conversion
- Verification of Type safety, according to CTS
- Stack Walker
- Memory Management and Garbage Collection
- Profiling and Debugging
- Co-Instance Execution
- Unmanaged Code

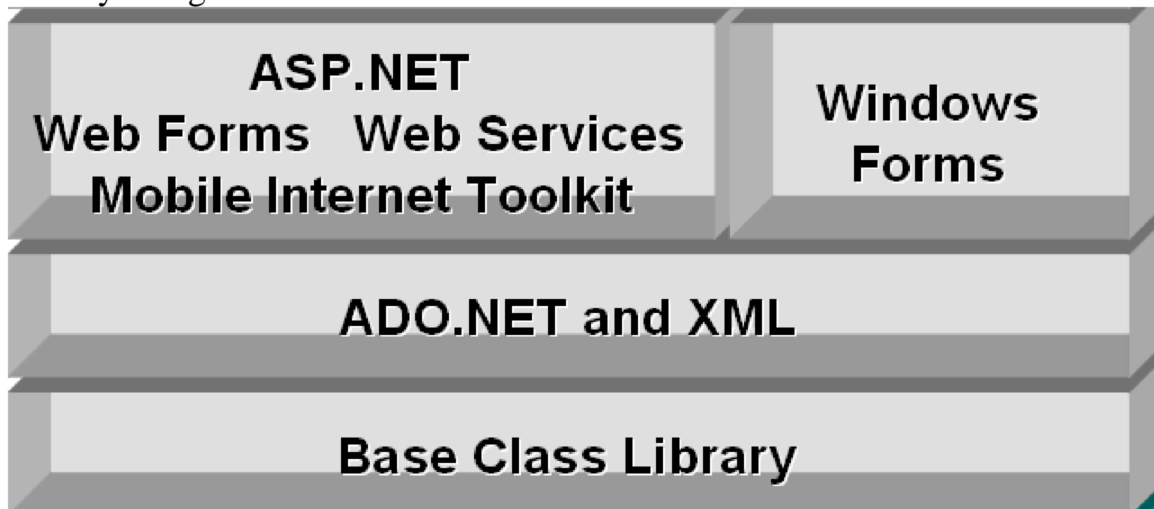
1.3 The .NET Framework Class Library:

The .Net Class Framework provides developers with object-oriented, extensible classes, interfaces and types for accessing system functionality. The Class Framework is organized into hierarchical libraries of classes that may be used consistently across any .Net compliant language.

The .Net Class Framework overcomes these limitations in the following ways:

- **Namespaces:** Classes, Interfaces and Types are organized into hierarchical structures called namespaces which group related classes and keep groups of classes distinct.
- **Unified Programming Framework:** .Net provides a Common Type System that standardizes data types across the Framework, Which puts all languages on an equal footing in terms of what data types they can communicate with. There are some differences between languages but in general they are all able to access the same classes.
- **Object-Oriented:** The Class Framework provides extensible classes that may be manipulated using standard object oriented operations including inheritance, method overriding and polymorphism.

The figure below shows the organization of the .Net Framework Class Library at high level:



The **System** namespace is the root namespace for all other namespaces in the .NET Framework.

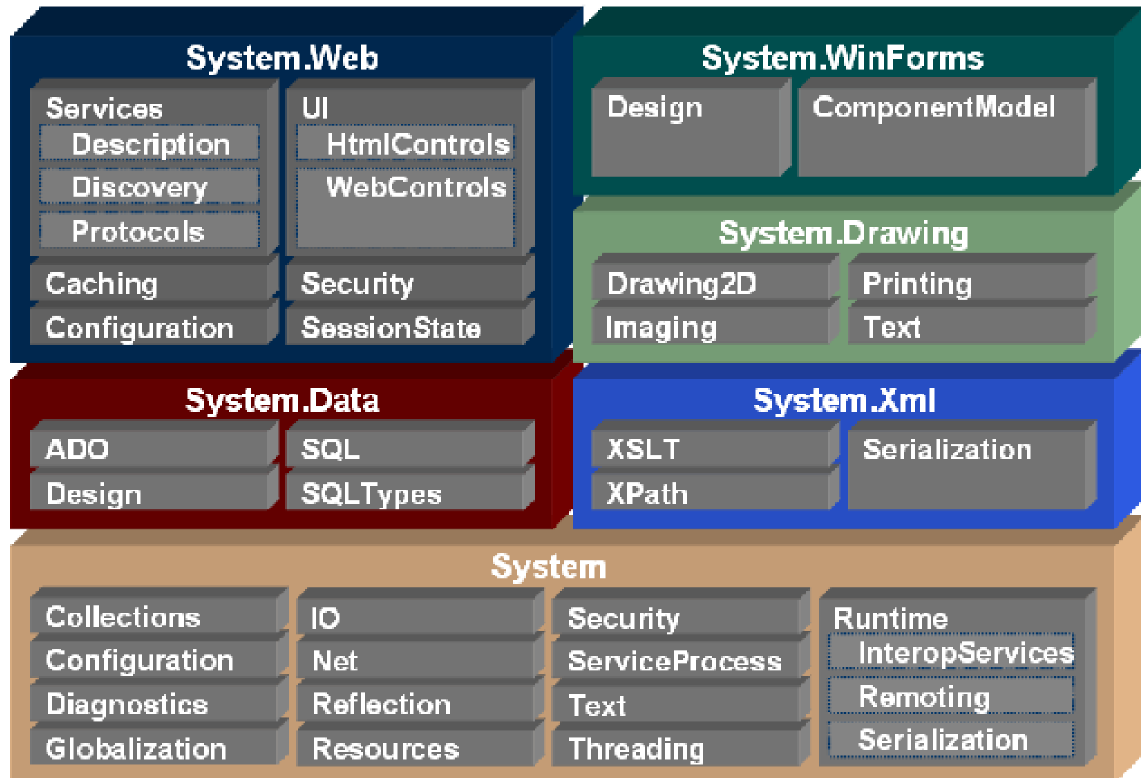
The important classes in the System namespace are:

- **System.Object:** All classes in the .NET Framework inheri this class. This class ensures that every Framework class implements a basic standard interface.
- **System.Exception:** Contains classes that standardize Exception handling

The features in the figure above translate to namespaces in the class library:

- **ADO.NET** provides the System.Data namespace for data access classes
- **ASP.NET** provides the System.Web namespace for ASP.NET code, including control and classes that support Web Services

- **XML.NET** provides the System.Xml namespace for XML classes. It contains other namespaces such as XPath, XSLT and Serialization namespaces
- **Windows Forms** provides the System.Windows.Forms namespace for classes that support windows forms control and functionality



The .NET Class Framework provides object-oriented access to a broad range of functionality, including direct system functionality.

What is the Global.asax used for?

The Global.asax (including the Global.asax.cs file) is used to implement application and session level events.

What is Web.Config File?

It is an optional XML File which stores configuration details for a specific asp.net web application.

Note: When you modify the settings in the Web.Config file, you do not need to restart the Web service for the modifications to take effect.. By default, the Web.Config file applies to all the pages in the current directory and its subdirectories.

Extra: You can use the tag to lock configuration settings in the Web.Config file so that they cannot be overridden by a Web.Config file located below it. You can use the allowOverride attribute to lock configuration settings. This attribute is especially valuable if you are hosting untrusted applications on your server.

What is Machine.config File?

The Machine.Config file, which specifies the settings that are global to a particular machine. This file is located at the following path:
\WINNT\Microsoft.NET\Framework\[Framework Version]\CONFIG\machine.config
As web.config file is used to configure one asp .net web application, same way Machine.config file is used to configure the application according to a particular machine. That is, configuration done in machine.config file is affected on any application that runs on a particular machine. Usually, this file is not altered and only web.config is used which configuring applications.
You can override settings in the Machine.Config file for all the applications in a particular Web site by placing a Web.Config file in the root directory of the Web site as follows:
\InetPub\wwwroot\Web.Config

Difference between Web.Config and Machine.Config File

Machine.Config:

i) This is automatically installed when you install Visual Studio. Net.
ii) This is also called machine level configuration file.
iii) Only one machine.config file exists on a server.
iv) This file is at the highest level in the configuration hierarchy.

Web.Config:

i) This is automatically created when you create an ASP.Net web application project.
ii) This is also called application level configuration file.
iii) This file inherits setting from the machine.config

2. Visual Basic .NET Programming

2.1 Working with toolbox Controls

Label, LinkLabel

Label

Labels are those controls that are used to display text in other parts of the application. They are based on the Control class.
Notable property of the label control is the text property which is used to set the text for the label.

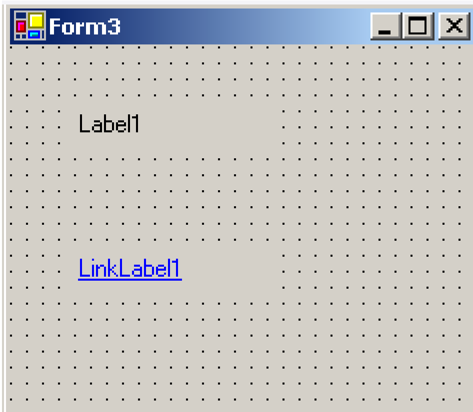
Label Event

The default event of Label is the Click event which looks like this in code:

```
Private Sub Label1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Label1.Click

End Sub
```

Creating a Label in Code

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load Dim Label1 As New Label()
    Label1.Text = "Label"
    Label1.Location = New Point(135, 70)
    Label1.Size = New Size(30, 30)
    Me.Controls.Add(Label1)
End Sub
```



LinkLabel

LinkLabel is similar to a Label but they display a hyperlink. Even multiple hyperlinks can be specified in the text of the control and each hyperlink can perform a different task within the application. They are based on the Label class which is based on the Control class.
Notable properties of the LinkLabel control are the ActiveLinkColor, LinkColor and LinkVisited which are used to set the link color.

LinkLabel Event

The default event of LinkLabel is the LinkClicked event which looks like this in code:

```
Private Sub LinkLabel1_LinkClicked(ByVal sender As System.Object, _
ByVal e As System.Windows.Forms.LinkLabelLinkClickedEventArgs)_
Handles LinkLabel1.LinkClicked

End Sub
```

Working with LinkLabel

Drag a LinkLabel (LinkLabel1) onto the form. When we click this LinkLabel it will take us to "**www.startvbdotnet.com**". The code for that looks like this:

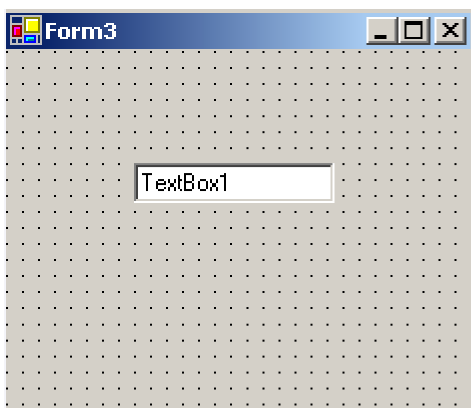
```
Private Sub LinkLabel1_LinkClicked(ByVal sender As System.Object, ByVal_
e As System.Windows.Forms.LinkLabelLinkClickedEventArgs)_
Handles LinkLabel1.LinkClicked
    System.Diagnostics.Process.Start("www.startvbdotnet.com")
    'using the start method of system.diagnostics.process class
     'process class gives access to local and remote processes
End Sub
```

Creating a LinkLabel in Code

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        Dim LinkLabel1 As New LinkLabel()
        LinkLabel1.Text = "Label"
        LinkLabel1.Location = New Point(135, 70)
        LinkLabel1.Size = New Size(30, 30)
        Me.Controls.Add(LinkLabel1)
End Sub
```

TextBox Control

Windows users should be familiar with textboxes. This control looks like a box and accepts input from the user. The TextBox is based on the TextBoxBase class which is based on the Control class. TextBoxes are used to accept input from the user or used to display text. By default we can enter up to 2048 characters in a TextBox but if the Multiline property is set to True we can enter up to 32KB of text. The image below displays a Textbox.



Some Notable Properties:

Some important properties in the Behavior section of the Properties Window for TextBoxes.

Enabled: Default value is True. To disable, set the property to False.
Multiline: Setting this property to True makes the TextBox multiline which allows to accept multiple lines of text. Default value is False.

PasswordChar: Used to set the password character. The text displayed in the TextBox will be the character set by the user. Say, if you enter *, the text that is entered in the TextBox is displayed as *.
ReadOnly: Makes this TextBox readonly. It doesn't allow to enter any text.
Visible: Default value is True. To hide it set the property to False.
Important properties in the Appearance section
TextAlign: Allows to align the text from three possible options. The default value is left and you can set the alignment of text to right or center.
Scrollbars: Allows to add a scrollbar to a Textbox. Very useful when the TextBox is multiline. You have four options with this property. Options are are None, Horizontal, Vertical and Both. Depending on the size of the TextBox anyone of those can be used.

TextBox Event

The default event of the TextBox is the TextChanged Event which looks like this in code:

```
Private Sub TextBox1_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TextBox1.TextChanged

End Sub
```

Working With TextBoxes

Lets work with some examples to understand TextBoxes.
Drag two TextBoxes (TextBox1, TextBox2) and a Button (Button1) from the toolbox.
Code to Display some text in the TextBox
We want to display some text, say, "Welcome to TextBoxes", in TextBox1 when the Button is clicked. The code looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles Button1.Click
        TextBox1.Text = "Welcome to TextBoxes"
End Sub
```

Code to Work with PassWord Character
Set the PasswordChar property of TextBox2 to *. Setting that will make the text entered in TextBox2 to be displayed as *. We want to display what is entered in TextBox2 in TextBox1. The code for that looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles Button1.Click
        TextBox1.Text = TextBox2.Text
End Sub
```

When you run the program and enter some text in TextBox2, text will be displayed as *. When you click the Button, the text you entered in TextBox2 will be displayed as plain text in TextBox1.

Code to Validate User Input

We can make sure that a TextBox can accept only characters or numbers which can restrict accidental operations. For example, adding two numbers of the form 27+2J cannot return anything. To avoid such kind of operations we use the KeyPress event of the TextBox.

Code that allows you to enter only double digits in a TextBox looks like this:

```
Private Sub TextBox1_KeyPress(ByVal sender As Object,ByVal e As_
System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
        If(e.KeyChar < "10" Or e.KeyChar > "100") Then
                MessageBox.Show("Enter Double Digits")
        End If
End Sub
```

Creating a TextBox in Code

```
Public Class Form1 Inherits System.Windows.Forms.Form
  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As_
  System.EventArgs) Handles MyBase.Load
       Dim TextBox1 as New TextBox()
       TextBox1.Text="Hello Mate"
       TextBox1.Location=New Point(100,50)
       TextBox1.Size=New Size(75,23)
       Me.Controls.Add(TextBox1)
  End Sub
End Class
```

Button Control
One of the most popular control in Visual Basic is the Button Control (previously Command Control). They are the controls which we click and release to perform some action. Buttons are used mostly for handling events in code, say, for sending data entered in the form to the database and so on. The default event of the Button is the Click event and the Button class is based on the ButtonBase class which is based on the Control class.
Button Event

The default event of the Button is the Click event. When a Button is clicked it responds with the Click Event. The Click event of Button looks like this in code:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles Button1.Click
        'You place the code here to perform action when Button is clicked
End Sub
```

Working with Buttons

Well, it's time to work with Buttons. Drag a Button from the toolbox onto the Form. The default text on the Button is Button1. Click on Button1 and select it's properties by pressing F4 on the keyboard or by selecting View->Properties Window from the main menu. That displays the Properties for Button1.

Important Properties of Button1 from Properties Window:

Appearance

Appearance section of the properties window allows us to make changes to the appearance of the Button. With the help of BackColor and Background Image properties we can set a background color and a background image to the button. We set the font color and font style for the text that appears on button with ForeColor and the Font property. We change the appearance style of the button with the FlatStyle property. We can change the text that appears on button with the Text property and with the TextAlign property we can set where on the button the text should appear from a predefined set of options.

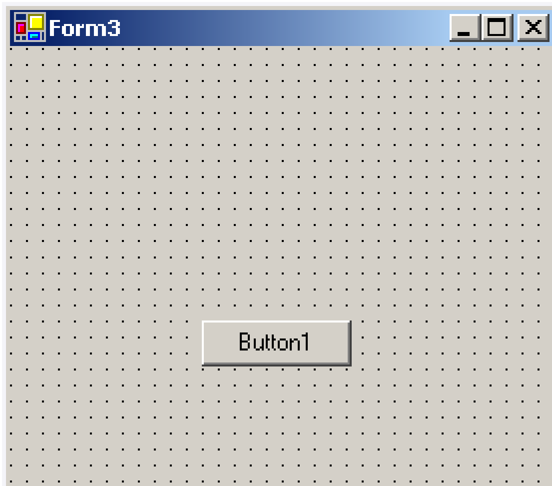
Behavior

Notable Behavior properties of the Button are the Enabled and Visible properties. The Enabled property is set to True by default which makes the button enabled and setting it's property to False makes the button Disabled. With the Visible property we can make the Button Visible or Invisible. The default value is set to True and to make the button Invisible set it's property to False.

Layout

Layout properties are about the look of the Button. Note the Dock property here. A control can be docked to one edge of its parent container or can be docked to all edges and fill the parent container. The default value is set to none. If you want to dock the control towards the left, right, top, bottom and center you can do that by selecting from the button like image this property displays. With the Location property you can change the location of the button. With the Size property you can set the size of the button. Apart from the Dock property you can set it's size and location by moving and stretching the Button on the form itself.

Below is the image of a Button.



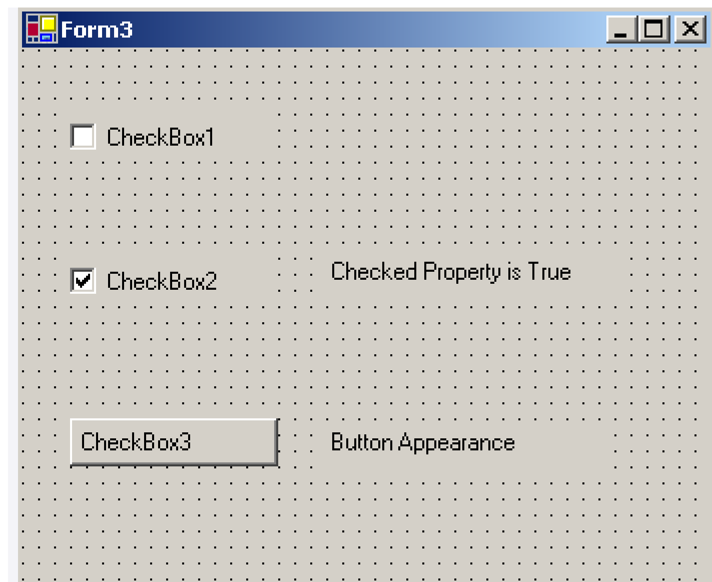
Creating a Button in Code

Below is the code to create a button.

```
Public Class Form1 Inherits System.Windows.Forms.Form
  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e_
  As System.EventArgs) Handles_ MyBase.Load
      Dim Button1 as New Button() 'declaring the button, Button1
      Button1.Text="Creating a Button" 'setting the text to be displayed on the Button
      Button1.Location=New Point(100,50)
      'setting the location for the Button where it should be created
      Button1.Size=New Size(75,23) 'setting the size of the Button
      Me.Controls.Add(Button1) 'adding the Button that is created to the form
      'the Me keyword is used to refer to the current object, in this case the Form
  End Sub
End Class
```

CheckBox

CheckBoxes are those controls which gives us an option to select, say, Yes/No or True/False. A checkbox is clicked to select and clicked again to deselect some option. When a checkbox is selected a check (a tick mark) appears indicating a selection. The CheckBox control is based on the TextBoxBase class which is based on the Control class. Below is the image of a Checkbox.



Notable Properties

Important properties of the CheckBox in the Appearance section of the properties window are:

Appearance: Default value is Normal. Set the value to Button if you want the CheckBox to be displayed as a Button.
BackgroundImage: Used to set a background image for the checkbox.
CheckAlign: Used to set the alignment for the CheckBox from a predefined list.
Checked: Default value is False, set it to True if you want the CheckBox to be displayed as checked.
CheckState: Default value is Unchecked. Set it to True if you want a check to appear. When set to Indeterminate it displays a check in gray background.
FlatStyle: Default value is Standard. Select the value from a predefined list to set the style of the checkbox.

Important property in the Behavior section of the properties window is the ThreeState property which is set to False by default. Set it to True to specify if the Checkbox can allow three check states than two.

CheckBox Event
The default event of the CheckBox is the CheckedChange event which looks like this in code:

```
Private Sub CheckBox1_CheckedChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles CheckBox1.CheckedChanged

End Sub
```

Working with CheckBoxes

Lets work with an example. Drag a CheckBox (CheckBox1), TextBox (TextBox1) and a Button (Button1) from the Toolbox.

Code to display some text when the Checkbox is checked

```
Private Sub CheckBox1_CheckedChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles CheckBox1.CheckedChanged
            TextBox1.Text = "CheckBox Checked"
End Sub
```

Code to check a CheckBox's state

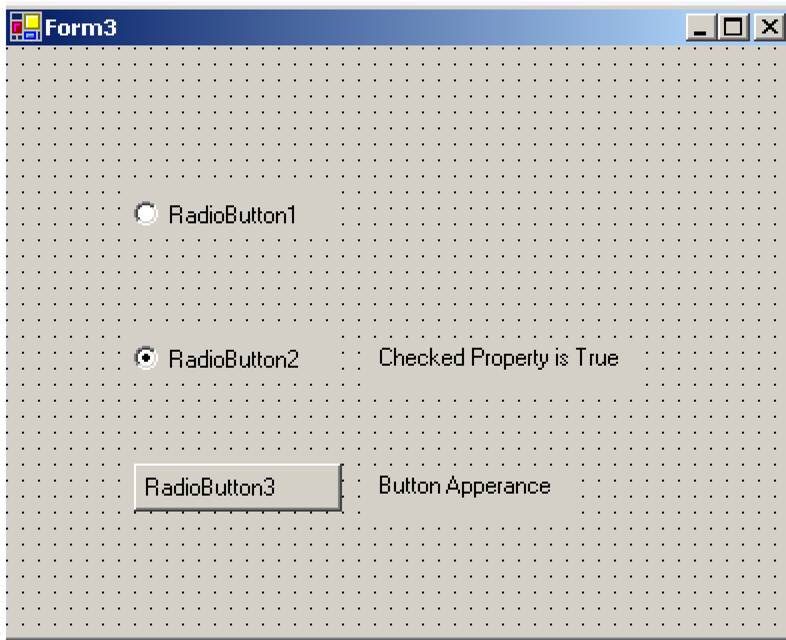
```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles Button1.Click
  If CheckBox1.Checked = True Then
     TextBox1.Text = "Checked"
  Else
     TextBox1.Text = "UnChecked"
  End If
End Sub
```

Creating a CheckBox in Code

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles_ MyBase.Load
     Dim CheckBox1 As New CheckBox()
     CheckBox1.Text = "Checkbox1"
     CheckBox1.Location = New Point(100, 50)
     CheckBox1.Size = New Size(95, 45)
     Me.Controls.Add(CheckBox1)
End Sub
```

RadioButton

RadioButtons are similar to CheckBoxes but RadioButtons are displayed as rounded instead of boxed as with a checkbox. Like CheckBoxes, RadioButtons are used to select and deselect options but they allow us to choose from mutually exclusive options. The RadioButton control is based on the ButtonBase class which is based on the Control class. A major difference between CheckBoxes and RadioButtons is, RadioButtons are mostly used together in a group. Below is the image of a RadioButton.



Important properties of the RadioButton in the Appearance section of the properties window are:

Appearance: Default value is Normal. Set the value to Button if you want the RadioButton to be displayed as a Button.
BackgroundImage: Used to set a background image for the RadioButton.
CheckAlign: Used to set the alignment for the RadioButton from a predefined list.
Checked: Default value is False, set it to True if you want the RadioButton to be displayed as checked.
FlatStyle: Default value is Standard. Select the value from a predefined list to set the style of the RadioButton.

RadioButton Event

The default event of the RadioButton is the CheckedChange event which looks like this in code:

```
Private Sub RadioButton1_CheckedChanged(ByVal sender As System.Object,_
ByVal e As System.EventArgs) Handles RadioButton1.CheckedChanged

End Sub
```

Working with Examples

Drag a RadioButton (RadioButton1), TextBox (TextBox1) and a Button (Button1) from the Toolbox.

Code to display some text when the RadioButton is selected

```
Private Sub RadioButton1_CheckedChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles RadioButton1.CheckedChanged
    TextBox1.Text = "RadioButton Selected"
End Sub
```

Code to check a RadioButton's state

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As_
System.EventArgs) Handles Button1.Click
If RadioButton1.Checked = True Then
TextBox1.Text = "Selected"
Else
TextBox1.Text = "Not Selected"
End If
End Sub
```

Creating a RadioButton in Code

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
As_
System.EventArgs) Handles_ MyBase.Load
Dim RadioButton1 As New RadioButton()
RadioButton1.Text = "RadioButton1"
RadioButton1.Location = New Point(120,60)
RadioButton1.Size = New Size(100, 50)
Me.Controls.Add(RadioButton1)
End Sub
```

Date TimePicker, Month Calendar, Splitter

Date TimePicker

Date TimePicker allows us to select date and time. Date TimePicker is based on the control class. When we click on the drop-down arrow on this control it displays a month calendar from which we can make selections. When we make a selection that selection appears in the textbox part of the Date TimePicker. The image below displays the Date TimePicker.

Notable Properties of Date TimePicker

The Format property in the Appearance section is used to select the format of the date and time selected. Default value is long which displays the date in long format. Other values include short, time and custom

Behavior Section

The CustomFormat property allows us to set the format for date and time depending on what we like. To use the CustomFormat property we need to set the Format property to Custom.

The MaxDate Property allows us to set the maximum date we want the Date TimePicker to hold. Default MaxDate value set by the software is 12/31/9998.

The MinDate Property allows us to set the minimum date we want the Date TimePicker to hold. Default MinDate value set by the software is 1/1/1753.

MonthCalendar

The MonthCalendar control allows us to select date. The difference between a Date TimePicker and MonthCalendar is, in MonthCalendar we select the date visually and in Date TimePicker when we want to make a selection we click on the drop-down arrow and select the date from the MonthCalendar which is displayed. The image below displays a MonthCalendar control.

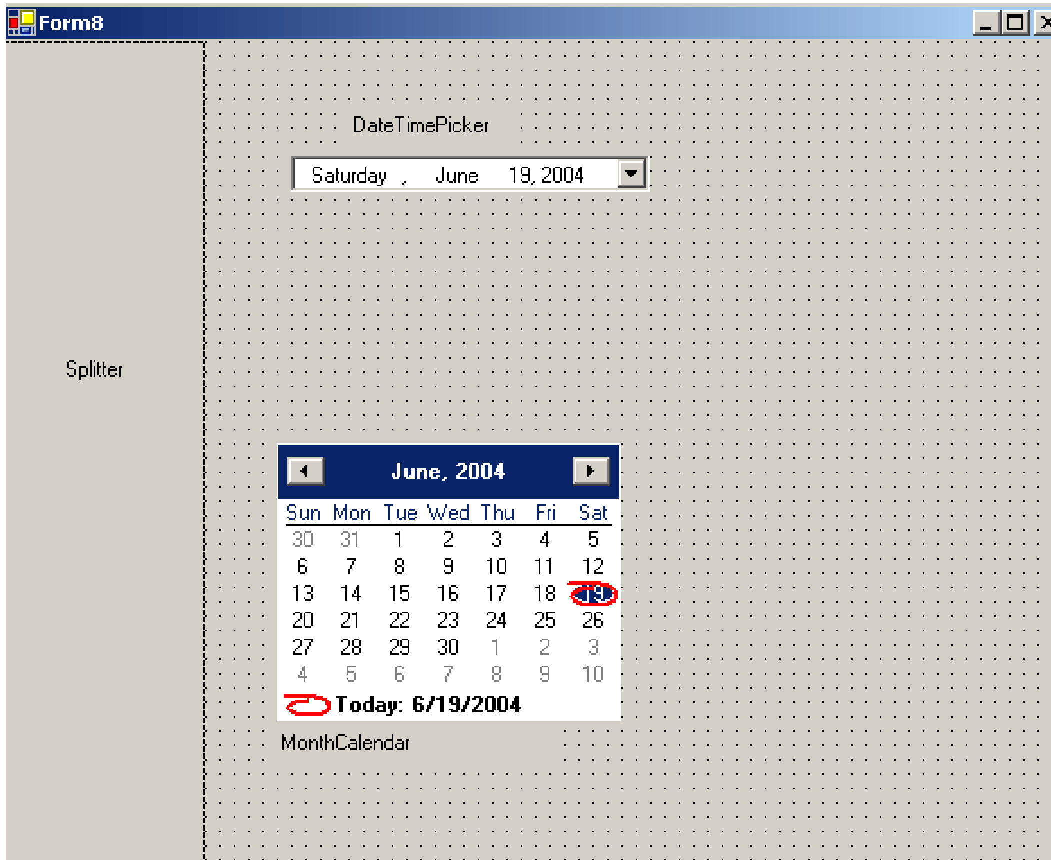
Notable Behavior properties of MonthCalendar

FirstDayOfWeek: Default value is Default which means that the week starts with Sunday as the first day and Saturday as last. You can set the first day of the week depending upon your choice by selecting from the predefined list with this property.

ShowToday: Default value is set to True which displays the current date at the bottom of the Calendar. Setting it to False will hide it.

ShowTodayCircle: Default value is set to True which displays a red circle on the current date. Setting it to False will make the circle disappear.

ShowWeekNumber: Default is False. Setting it to True will display the week number of the current week in the 52 week year. That will be displayed towards the left side of the control.



Splitter

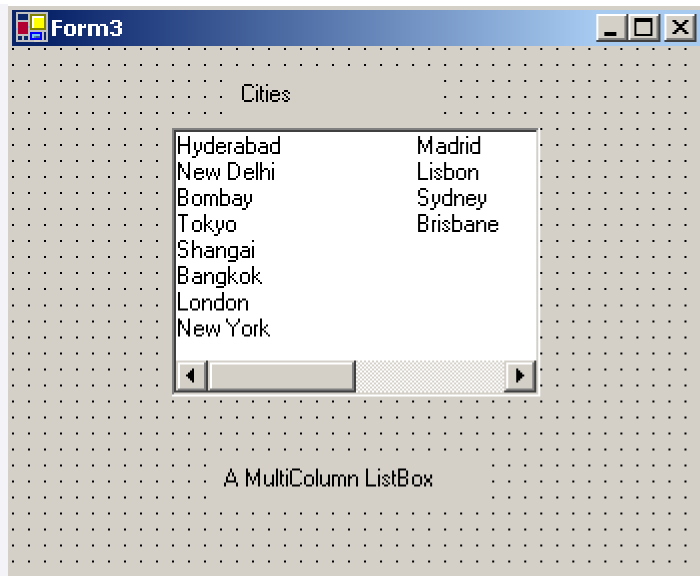
The Splitter control is used to resize other controls. The main purpose of Splitter control is to save space on the form. Once when we finish working with a particular control we can move it away from it's position or resize them with Splitter control. The Splitter control is invisible when we run the application but when the mouse is over it, the mouse cursor changes indicating that it's a Splitter control and it can be resized. This control can be very useful when we are working with controls both at design time and run time (which are not visible at design time). The Splitter control is based on the Control class.

Working with Splitter Control

To work with a Splitter Control we need to make sure that the other control with which this control works is docked towards the same side of the container. Let's do that with an example. Assume that we have a TextBox on the form. Drag a Splitter control onto the form. Set the TextBox's dock property to left. If we want to resize the TextBox once we finish using it set the Splitter's dock property to left (both the controls should be docked towards the same end). When the program is executed and when you pass the mouse over the Splitter control it allows us to resize the TextBox allowing us to move it away from it's current position.

ListBox

The ListBox control displays a list of items from which we can make a selection. We can select one or more than one of the items from the list. The ListBox control is based on the ListControl class which is based on the Control class. The image below displays a ListBox.



Notable Properties of the ListBox

In the Behavior Section

HorizontalScrollbar: Displays a horizontal scrollbar to the ListBox. Works when the ListBox has MultipleColumns.
MultiColumn: The default value is set to False. Set it to True if you want the list box to display multiple columns.
ScrollAlwaysVisible: Default value is set to False. Setting it to True will display both Vertical and Horizontal scrollbar always.
SelectionMode: Default value is set to one. Select option None if you do not any item to be selected. Select it to MultiSimple if you want multiple items to be selected. Setting it to MultiExtended allows you to select multiple items with the help of Shift, Control and arrow keys on the keyboard.
Sorted: Default value is set to False. Set it to True if you want the items displayed in the ListBox to be sorted by alphabetical order.

In the Data Section

Notable property in the Data section of the Properties window is the Items property. The Items property allows us to add the items we want to be displayed in the list box. Doing so is simple, click on the ellipses to open the String Collection Editor window and start

entering what you want to be displayed in the ListBox. After entering the items click OK and doing that adds all the items to the ListBox.

ListBox Event

The default event of ListBox is the SelectedIndexChanged which looks like this in code:

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles
ListBox1.SelectedIndexChanged

End Sub
```

Working with ListBoxes

Drag a TextBox and a ListBox control to the form and add some items to the ListBox with it's items property.

Referring to Items in the ListBox

Items in a ListBox are referred by index. When items are added to the ListBox they are assigned an index. The first item in the ListBox always has an index of 0 the next 1 and so on.

Code to display the index of an item

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles
ListBox1.SelectedIndexChanged
TextBox1.Text = ListBox1.SelectedIndex
'using the selected index property of the list box to select the index
End Sub
```

When you run the code and select an item from the ListBox, it's index is displayed in the textbox.

Counting the number of Items in a ListBox

Add a Button to the form and place the following code in it's click event.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _
As System.EventArgs) Handles Button1.Click
TextBox1.Text = ListBox1.Items.Count
```

```
'counting the number of items in the ListBox with the Items.Count
End Sub
```

When you run the code and click the Button it will display the number of items available in the ListBox.

Code to display the item selected from ListBox in a TextBox

```
Private Sub ListBox1_SelectedIndexChanged(ByVal sender As
System.Object,_
ByVal e As System.EventArgs) Handles
ListBox1.SelectedIndexChanged
TextBox1.Text = ListBox1.SelectedItem
'using the selected item property
End Sub
```

When you run the code and click an item in the ListBox that item will be displayed in the TextBox.

Code to Remove items from a ListBox

You can remove all items or one particular item from the list box.

Code to remove a particular item

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _
As System.EventArgs) Handles Button1.Click
ListBox1.Items.RemoveAt(4)
'removing an item by specifying it's index
End Sub
```

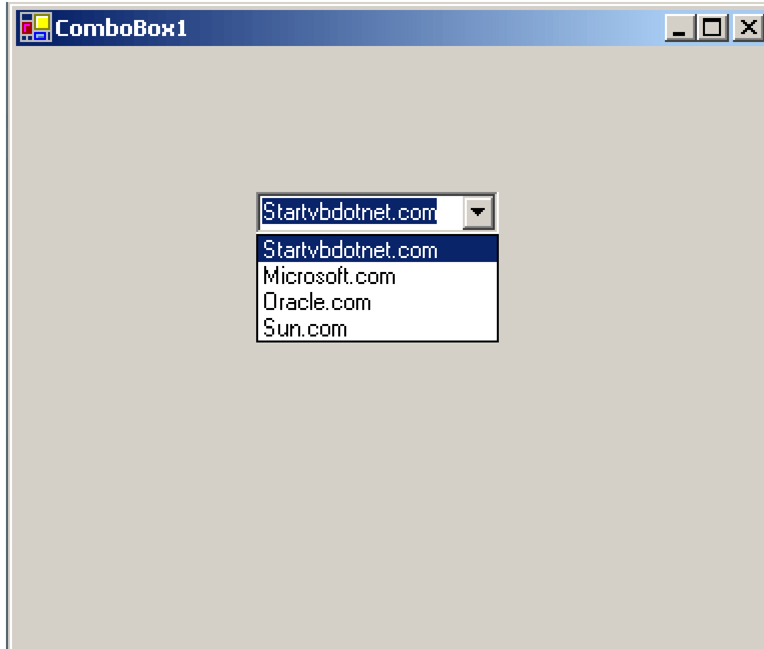
Code to Remove all items

```
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
ListBox1.Items.Clear()
'using the clear method to clear the list box
End Sub
```

ComboBox

ComboBox is a combination of a TextBox and a ListBox. The ComboBox displays an editing field (TextBox) combined with a ListBox allowing us to select from the list or to enter new text. ComboBox displays data in a drop-down style format. The ComboBox

class is derived from the ListBox class. Below is the Image of a ComboBox.



Notable properties of the ComboBox

The DropDownStyle property in the Appearance section of the properties window allows us to set the look of the ComboBox. The default value is set to DropDown which means that the ComboBox displays the Text set by it's Text property in the Textbox and displays it's items in the DropDownListBox below. Setting it to simple makes the ComboBox to be displayed with a TextBox and the list box which doesn't drop down. Setting it to DropDownList makes the ComboBox to make selection only from the drop down list and restricts you from entering any text in the textbox.

We can sort the ComboBox with it's Sorted property which is set to False by Default.

We can add items to the ComboBox with it's Items property.

ComboBox Event

The default event of ComboBox is SelectedIndexChanged which looks like this in code:

```
Private Sub ComboBox1_SelectedIndexChanged(ByVal sender As
System.Object,_
ByVal e As System.EventArgs) Handles
ComboBox1.SelectedIndexChanged

End Sub
```

Working with ComboBoxes

Drag a ComboBox and a TextBox control onto the form. To display the selection made in the ComboBox in the Textbox the code looks like this:

```
Private Sub ComboBox1_SelectedIndexChanged(ByVal sender As
System.Object,_
ByVal e As System.EventArgs) Handles
ComboBox1.SelectedIndexChanged
TextBox1.Text = ComboBox1.SelectedItem
'selecting the item from the ComboBox with selected item property
End Sub
```

Removing items from a ComboBox

You can remove all items or one particular item from the list box part of the ComboxBox. Code to remove a particular item by it's Index number looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As_
System.EventArgs) Handles Button1.Click
ComboBox1.Items.RemoveAt(4)
'removing an item by specifying it's index
End Sub
```

Code to remove all items from the ComboBox

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As_
System.EventArgs) Handles Button1.Click
ComboBox1.Items.Clear()
'using the clear method to clear the list box
End Sub
```

Panel, GroupBox, PictureBox

Panel

Panels are those controls which contain other controls, for example, a set of radio buttons, checkboxes, etc. Panels are similar to Groupboxes but the difference, Panels cannot display captions where as GroupBoxes can and Panels can have scrollbars where as GroupBoxes can't. If the Panel's Enabled property is set to False then the controls which the Panel contains are also disabled. Panels are based on the ScrollableControl class.

Notable property of the Panel control in the appearance section is the BorderStyle property. The default value of the BorderStyle property is set to None. You can select from the predefined list to change a Panels BorderStyle.
Notable property in the layout section is the AutoScroll property. Default value is set to False. Set it to True if you want a scrollbar with the Panel.

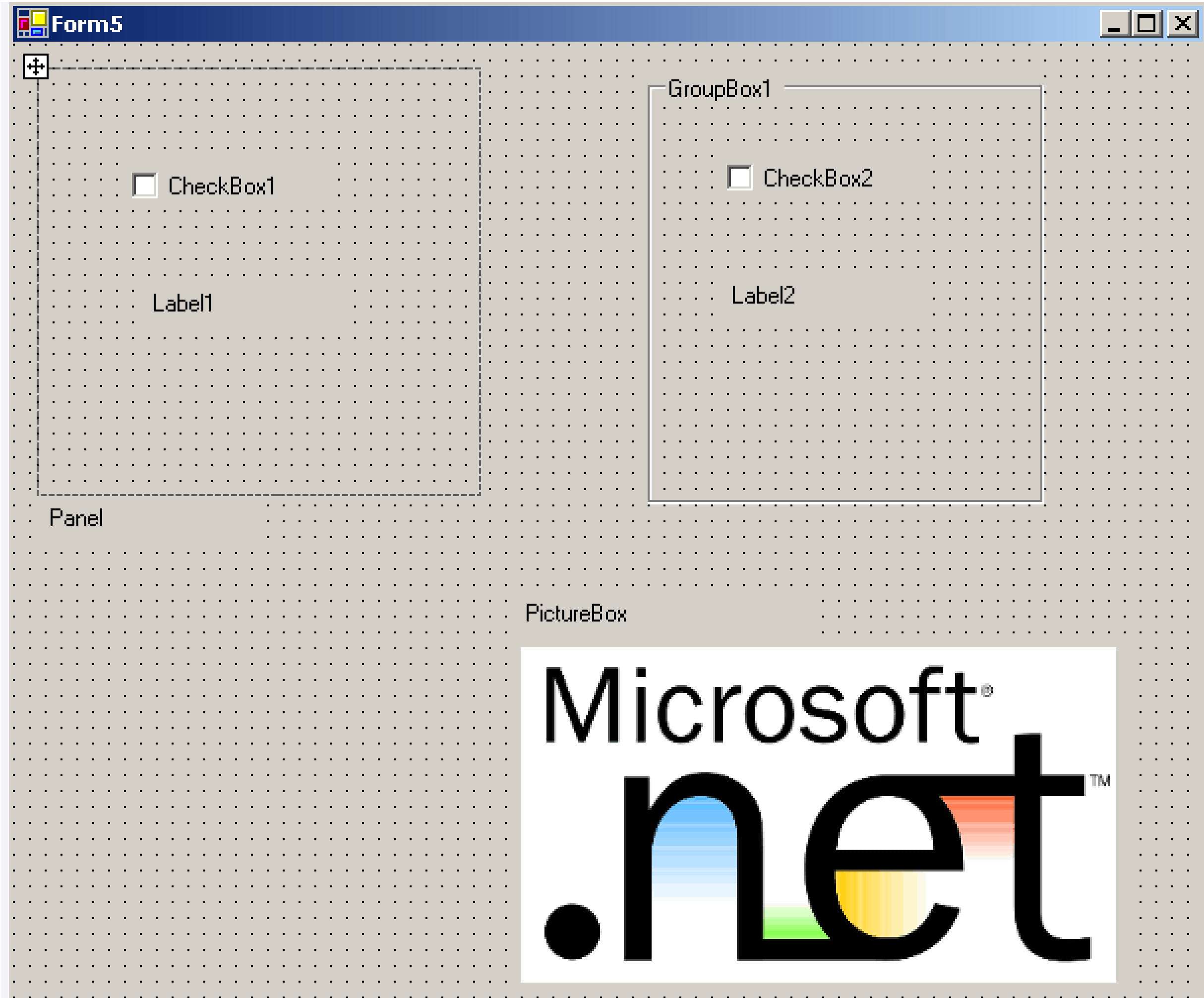
Adding Controls to a Panel

On a from drag a Panel (Panel1) from the toolbox. We want to place some controls, say, checkboxes on this Panel. Drag three checkboxes from the toolbox and place them on the Panel. When that is done all the checkboxes in the Panel are together as in a group but they can function independently.

<u>Creating a Panel and adding a Label and a CheckBox to it in Code</u>

```
Private Sub Form3_Load(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles MyBase.Load
Dim Panel1 As New Panel()
Dim CheckBox1 As New CheckBox()
Dim Label1 As New Label()
Panel1.Location = New Point(30, 60)
Panel1.Size = New Size(200, 264)
Panel1.BorderStyle = BorderStyle.Fixed3D
'setting the borderstyle of the panel
Me.Controls.Add(Panel1)
CheckBox1.Size = New Size(95, 45)
CheckBox1.Location = New Point(20, 30)
CheckBox1.Text = "Checkbox1"
Label1.Size = New Size(100, 50)
Label1.Location = New Point(20, 40)
Label1.Text = "CheckMe"
Panel1.Controls.Add(CheckBox1)
Panel1.Controls.Add(Label1)
'adding the label and checkbox to the panel
End Sub
```

The image below displays a panel.



GroupBox Control

As said above, Groupboxes are used to Group controls. GroupBoxes display a frame around them and also allows to display captions to them which is not possible with the Panel control. The GroupBox class is based on the Control class.

Creating a GroupBox and adding a Label and a CheckBox to it in Code

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles MyBase.Load
Dim GroupBox1 As New GroupBox()
Dim CheckBox1 As New CheckBox()
Dim Label1 As New Label()
GroupBox1.Location = New Point(30, 60)
GroupBox1.Size = New Size(200, 264)
GroupBox1.Text = "InGroupBox"
```

```
'setting the caption to the groupbox
Me.Controls.Add(GroupBox1)
CheckBox1.Size = New Size(95, 45)
CheckBox1.Location = New Point(20, 30)
CheckBox1.Text = "Checkbox1"
label1.Size = New Size(100, 50)
Label1.Location = New Point(20, 40)
Label1.Text = "CheckMe"
GroupBox1.Controls.Add(CheckBox1)
GroupBox1.Controls.Add(Label1)
'adding the label and checkbox to the groupbox
End Sub
```

PictureBox Control

PictureBoxes are used to display images on them. The images displayed can be anything varying from Bitmap, JPEG, GIF, PNG or any other image format files. The PictureBox control is based on the Control class.

Notable property of the PictureBox Control in the Appearance section of the properties window is the Image property which allows to add the image to be displayed on the PictureBox.

Adding Images to PictureBox

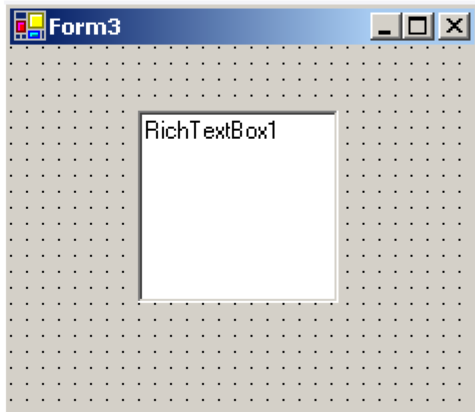
Images can be added to the PictureBox with the Image property from the Properties window or by following lines of code.

```
Private Sub Button1_Click(ByVal sender As System.Object,_
ByVal e As System.EventArgs) Handles Button1.Click
PictureBox1.Image = Image.FromFile("C:\sample.gif")
'loading the image into the picturebox using the FromFile method of
the image class
'assuming a GIF image named sample in C: drive
End Sub
```

RichTextBox

RichTextBoxes are similar to TextBoxes but they provide some advanced features over the standard TextBox. RichTextBox allows formatting the text, say adding colors, displaying particular font types and so on. The RichTextBox, like the TextBox is based on the TextBoxBase class which is based on the Control class. These RichTextBoxes came into existence because many word processors these days allow us to save text in a rich text format. With RichTextBoxes we can also create our own word processors. We have two options when accessing text in a RichTextBox, text and rtf (rich text format).

Text holds text in normal text and rtf holds text in rich text format. Image of a RichTextBox is shown below.



RichTextBox Event

The default event of RichtextBox is the TextChanged event which looks like this in code:

```
Private Sub RichTextBox1_TextChanged(ByVal sender As
System.Object, _
ByVal e As System.EventArgs) Handles RichTextBox1.TextChanged

End Sub
```

Code Samples

Code for creating bold and italic text in a RichTextBox

Drag a RichTextBox (RichTextBox1) and a Button (Button1) onto the form. Enter some text in RichTextBox1, say, "We are working with RichTextBoxes". Paste the following code in the click event of Button1. The following code will search for text we mention in code and sets it to be displayed as Bold or Italic based on what text is searched for.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles Button1.Click
RichTextBox1.SelectionStart = RichTextBox1.Find("are")
'using the Find method to find the text "are" and setting it's
'return property to SelectionStart which selects the text to format
Dim ifont As New Font(RichTextBox1.Font, FontStyle.Italic)
'creating a new font object to set the font style
RichTextBox1.SelectionFont = ifont
'assigning the value selected from the RichTextBox the font style
RichTextBox1.SelectionStart = RichTextBox1.Find("working")
Dim bfont As New Font(RichTextBox1.Font, FontStyle.Bold)
```

```
RichTextBox1.SelectionFont = bfont
End Sub
```

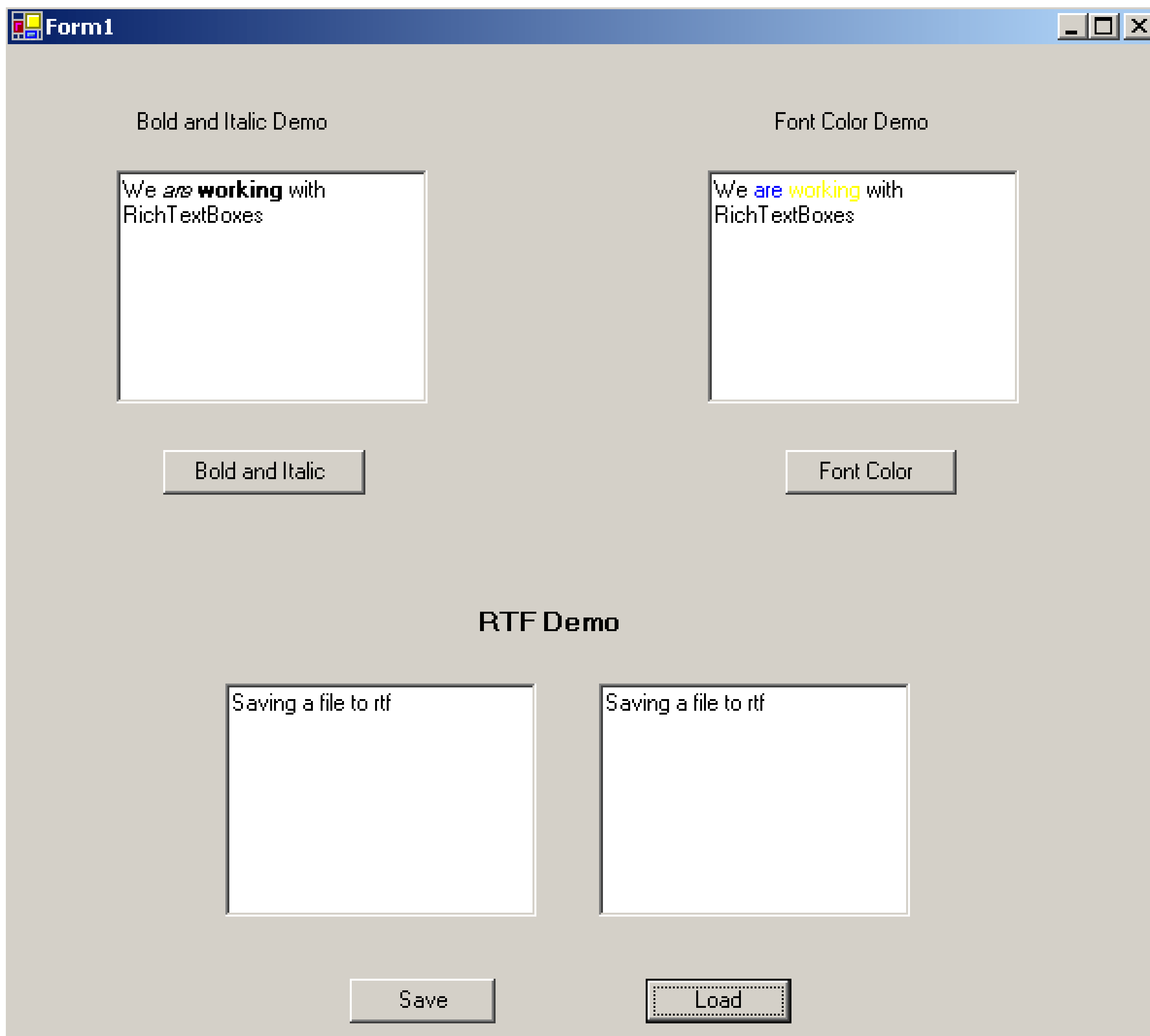
When you run the above code and click Button1, the text "*are*" is displayed in Italic and the text "**working**" is displayed in Bold font. The image below displays the output.

Code for Setting the Color of Text

Lets work with previous example. Code for setting the color for particular text looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As _
System.EventArgs) Handles Button1.Click
RichTextBox1.SelectionStart = RichTextBox1.Find("are")
'using the Find method to find the text "are" and setting it's return
'property to SelectionStart which selects the text
RichTextBox1.SelectionColor = Color.Blue
'setting the color for the selected text with SelectionColor property
RichTextBox1.SelectionStart = RichTextBox1.Find("working")
RichTextBox1.SelectionColor = Color.Yellow
End Sub
```

The output when the Button is Clicked is the text "are" being displayed in Blue and the text "working" in yellow as shown in the image below.



Code for Saving Files to RTF

Drag two RichTextBoxes and two Buttons (Save, Load) onto the form. When you enter some text in RichTextBox1 and click on Save button, the text from RichTextBox1 is saved into a rtf (rich text format) file. When you click on Load button the text from the rtf file is displayed into RichTextBox2. The code for that looks like this:

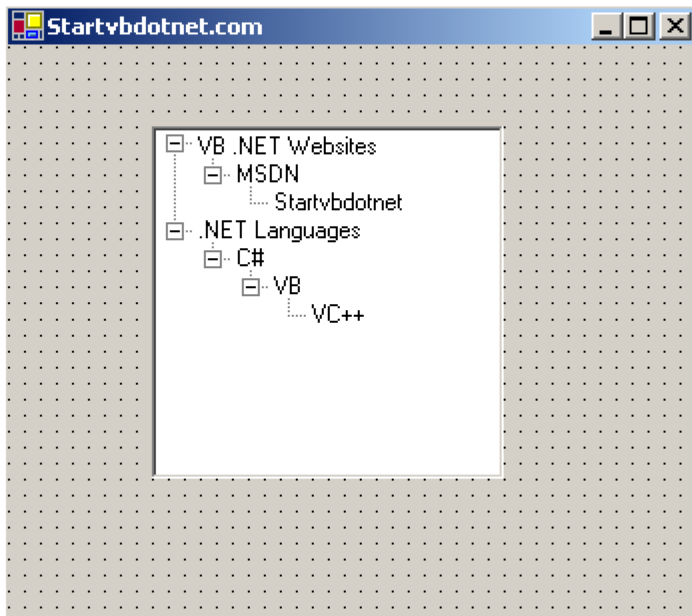
```
Private Sub Save_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles Save.Click
RichTextBox1.SaveFile("hello.rtf")
'using SaveFile method to save text in a rich text box to hard disk
End Sub

Private Sub Load_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles Load.Click
```

```
RichTextBox2.LoadFile("hello.rtf")
'using LoadFile method to read the saved file
End Sub
```

The files which we create using the SaveFile method are saved in the bin directory of the Windows Application. You can view output of the above said code in the image above.

TreeView

The tree view control is used to display a hierarchy of nodes (both parent, child). You can expand and collpase these nodes by clicking them. This control is similar to Windows Explorer which displays a tree view in it's left pane to list all the folders on the hard disk. Below is the image of a Tree View control.



Notable Properties of TreeView

Bounds: Gets the actual bound of the tree node
Checked: Gets/Sets whether the tree node is checked
FirstNode: Gets the first child tree node
FullPath: Gets the path from the root node to the current node
ImageIndex: Gets/Sets the image list index of the image displayed for a node
Index: Gets the location of the node in the node collection
IsEditing: Gets whether the node can be edited
IsExpaned: Gets whether the node is expaned
IsSelected: Gets whether the node is selected
LastNode: Gets the last child node

NextNode: Gets the next sibling node
NextVisibleNode: Gets the next visible node
NodeFont: Gets/Sets the font for nodes
Nodes: Gets the collection of nodes in the current node
Parent: Gets the parent node of the current node
PrevNode: Gets the previous sibling node
PrevVisibleNode: Gets the previous visible node
TreeView: Gets the node's parent tree view

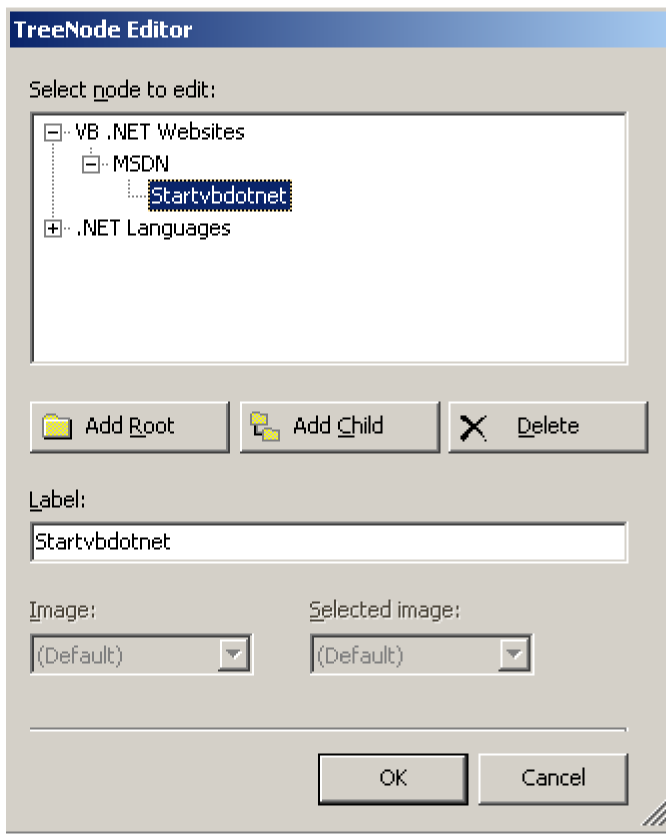
TreeView Event

Default event of the Tree View control is the AfterSelect event which looks like this in code:

```
Private Sub TreeView1_AfterSelect(ByVal sender As System.Object,
ByVal e As_
System.Windows.Forms.TreeViewEventArgs) Handles
TreeView1.AfterSelect

End Sub
```

Working with Tree Views

Drag a Tree View control on to a form and to add nodes to it select the nodes property in the properties window, which displays the TreeNode editor as shown below.



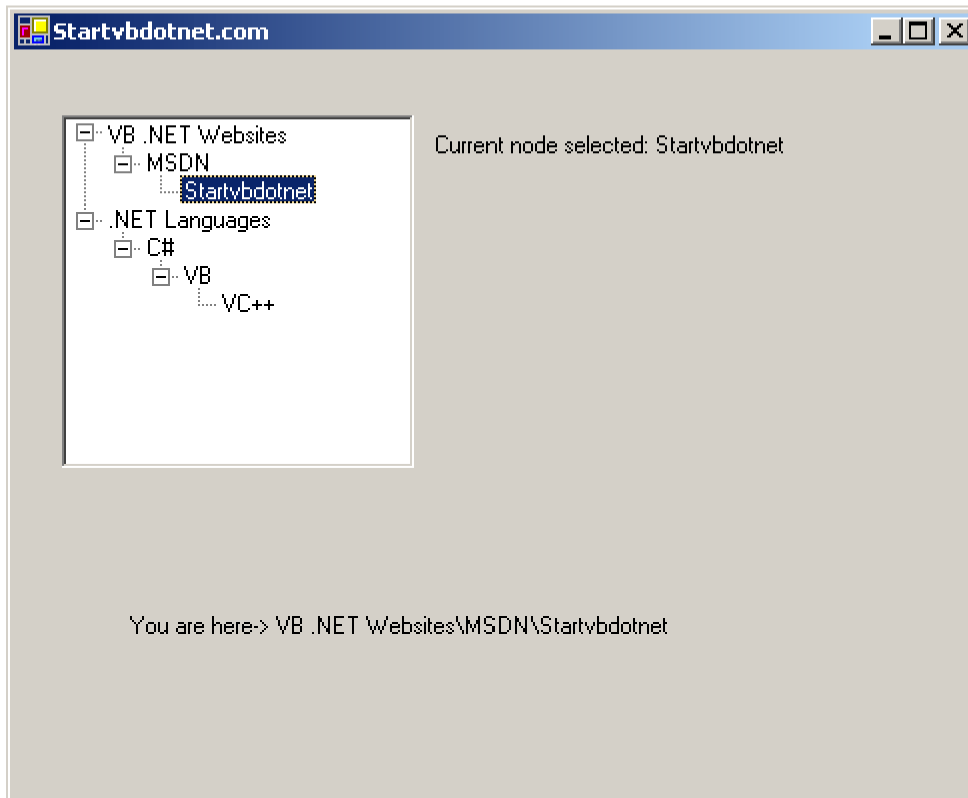
To start adding nodes, you should click the Add Root button, which adds a top-level node. To add child nodes to that node, you should select that node and use the Add Child button. To set text for a node, select the node and set it's text in the textbox as shown in the image above.

Assuming you added some nodes to the tree view, drag two Labels (Label1, Label2) from the toolbox on to the form. The following code displays the node you select on Label2 and the path to that node on Label1. The code looks like this:

```
Public Class Form12 Inherits System.Windows.Forms.Form
#Region " Windows Form Designer generated code "

#End Region

Private Sub TreeView1_AfterSelect(ByVal sender As System.Object,
ByVal e As_
System.Windows.Forms.TreeViewEventArgs) Handles
TreeView1.AfterSelect
Label1.Text = "You are here->" & " " & e.Node.FullPath
'displaying the path of the selected node
Label2.Text = "Current node selected:" & " " & e.Node.Text
'displaying the selected node
```

```
End Sub

End Class
```

The image below displays sample output from above code.



ToolTip, ErrorProvider

ToolTip

ToolTips are those small windows which display some text when the mouse is over a control giving a hint about what should be done with that control. ToolTip is not a control but a component which means that when we drag a ToolTip from the toolbox onto a form it will be displayed on the component tray. Tooltip is an Extender provider component which means that when you place an instance of a ToolTipProvider on a form, every control on that form receives a new property. This property can be viewed and set in the properties window where it appears as Tooltip on *n*, where *n* is the name of the ToolTipProvider.

To assign ToolTip's with controls we use it's SetToolTip method.

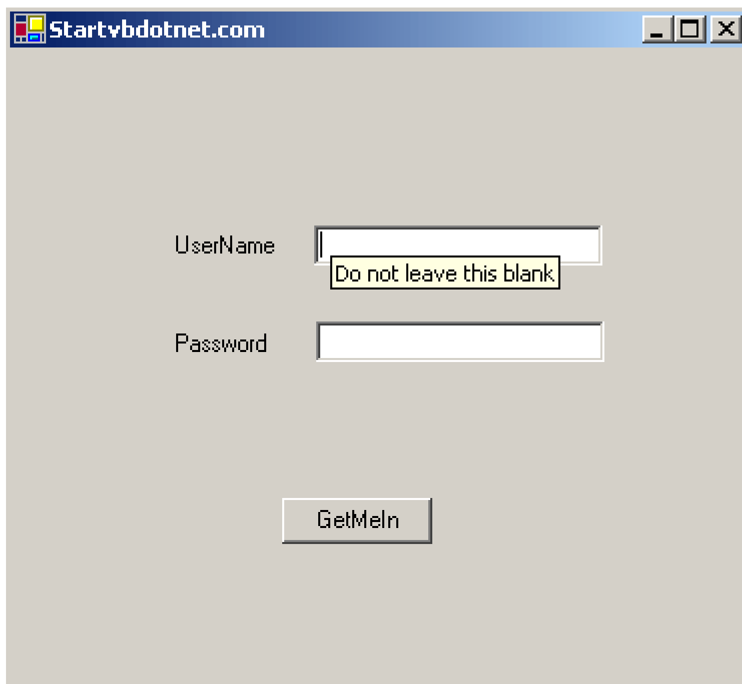
Notable property of the ToolTip is the Active property which is set to True by default and which allows the tool tip to be displayed.

Setting a ToolTip

Assume that we have a TextBox on the form and we want to display some text when your mouse is over the TextBox. Say the text that should appear is "Do not leave this blank". The code for that looks like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles MyBase.Load
ToolTip1.SetToolTip(TextBox1, "Do not leave this blank")
End Sub
```

The image below displays output from above code.



ErrorProvider Component

The ErrorProvider component provides an easy way to set validation errors. It allows us to set an error message for any control on the form when the input is not valid. When an error message is set, an icon indicating the error will appear next to the control and the error message is displayed as Tool Tip when the mouse is over the control.

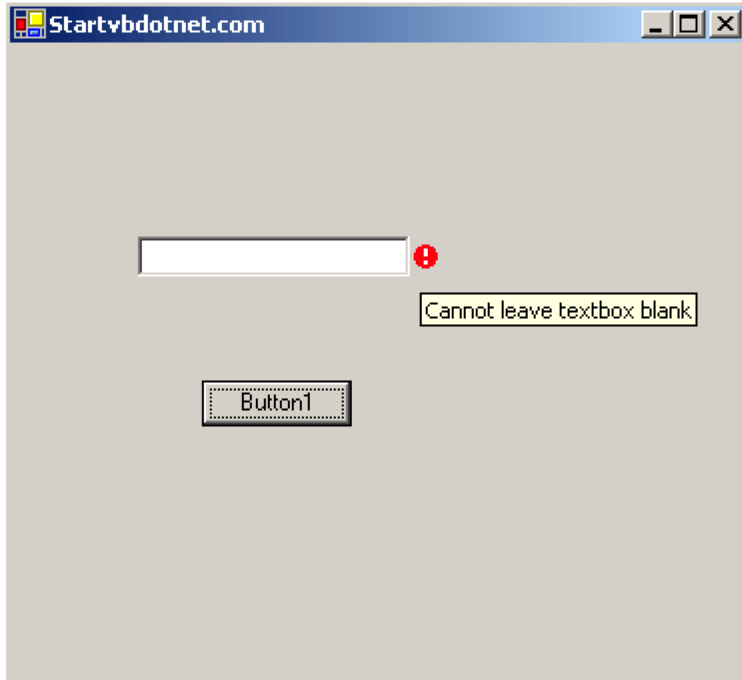
Notable property of ErrorProvider in the Appearance section is the Icon property which allows us to set an icon that should be displayed. Notable property in Behavior section is the BlinkRate property which allows to set the rate in milliseconds at which the icon blinks.

Displaying an Error

Let's work with an example. Assume we have a TextBox and a Button on a form. If the TextBox on the form is left blank and if the Button is clicked, an icon will be displayed next to the TextBox and the specified text will appear in the Tool Tip box when the mouse is over the control. The code for that looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e_
As System.EventArgs) Handles Button1.Click
If TextBox1.Text = "" Then
ErrorProvider1.SetError(TextBox1, "Cannot leave textbox blank")
Else
ErrorProvider1.SetError(TextBox1, "")
End If
End Sub
```

The image below displays output from above code.



Progress Bar

Create a New Project in VB.net. Drag a Progress bar control from tool box and place on form and now drag and drop four buttons on form having text I< << >> >I. this is simple interface for this purpose.Now write code on form load even of the form

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        ProgressBar1.Minimum = 0
        ProgressBar1.Maximum = 100
```

```
        ProgressBar1.Value = 0

End Sub
```

- Now write code on button click events.

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

        If ProgressBar1.Value < 100 Then
            ProgressBar1.Value += 5
        End If
End Sub

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click

        If ProgressBar1.Value > 0 Then
            ProgressBar1.Value -= 5
        End If
End Sub

Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button3.Click

        ProgressBar1.Value = 100
End Sub

Private Sub Button4_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button4.Click

        ProgressBar1.Value = 0
End Sub
```

This is code for buttons now we will see how we can work with progress bar with mouse wheel scrolling. Select MouseWheel event from form1 events.Now write simple code in this event:

```
Private Sub Form1_MouseWheel(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseWheel

        If e.Delta > -1 Then
            If ProgressBar1.Value < 100 Then
                ProgressBar1.Value += 5

            End If
        Else
            If ProgressBar1.Value > 0 Then
                ProgressBar1.Value -= 5

            End If
        End If

End Sub
```

Now in if condition you see that e.delta > -1 this is important for us when we scroll mouse one time control comes in this event and if we scroll wheel up side e.delta value will < 0 and if we scroll down side e.delta value > 0. Only remember these things in mind. Now see this condition ProgressBar1.Value < 100 we are handling the exception and also in ProgressBar1.Value > 0 because progressbar value should be in range of progressbar1.minimum and progressbar1.maximum.

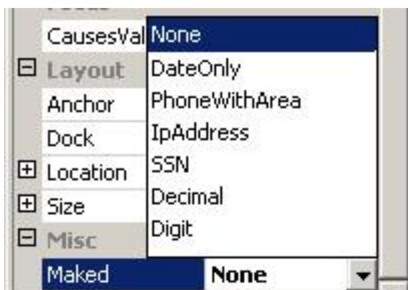
Masked Text Box

The TextBox control is the most used control in window program. It also cause a lot of problems either from QA or user, because the invalid data that were entered. Using masked control will solve these problems and save a lot of time for developer.

This masked intelligent user control enhances the function of TextBox control, which can mask the Date, IP Address, SSN, Phone number, digit, decimal and check the validation, automatically set delimit location.

The property Masked is set to None by default and the control works like a normal TextBox control.

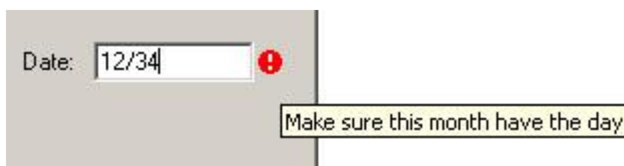
If setting the property to DateOnly, the control is masked to Date format.



Ex. When DateOnly is true

When user enter	Display
12	12/
124	12/04/
13	01/3
3	03/
34	03/04/
14	01/04/
1/	01/

Using the ErroProvider to handle the invalidate input:



Creating Control:

1. Start the Visual Studio.NET Windows Forms designer.
2. Select a new VB.net project by clicking New from the File menu.
3. Click Windows control library template on the templates.
4. Set the Name **MaskedTextBox**

NotifyIcon

Notify Icons display an icon in Windows System Tray. This is really useful for processes that run in the background and don't have their own interface. Since VB allows us to create Windows Services (services that run in the background and display control panels) now, we can use these notify icon's to associate funtionality to windows services. You can also use this icon to associate help with your application, launch another application or anything else which you think can be appropriate.

Notable properties of Notify Icon:

ContextMenu: Gets/Sets Context menu for the tray icon
Icon: Gets/Sets current icon
Text: Gets/Sets tooltip text that is displayed when the mouse hovers over the system tray
Visible: Gets/Sets if the icon is visible in the windows system tray

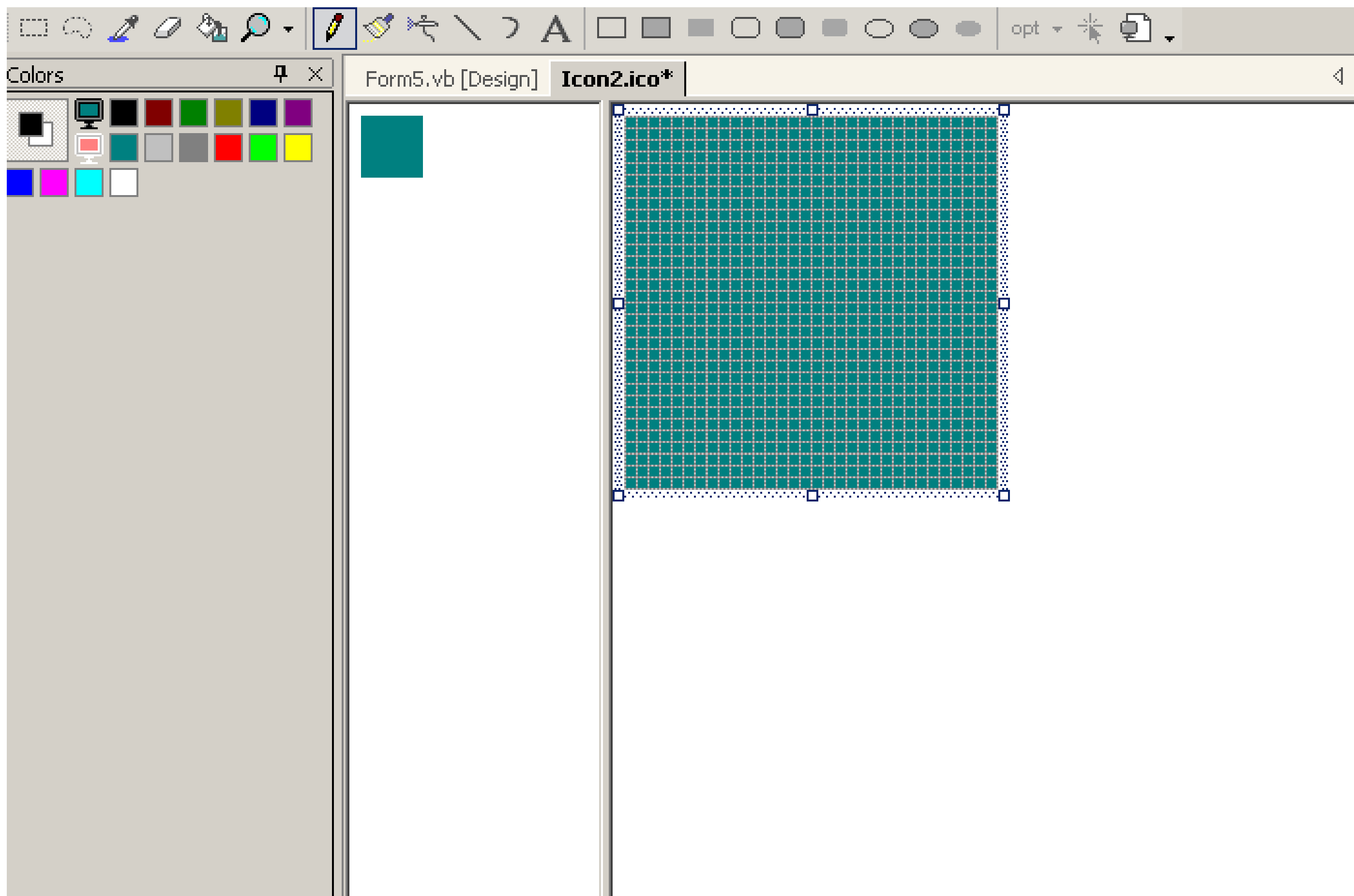
Notify Icon Event

The default event associated with Notify Icon is the MouseDown event which looks like this in code:

```
Private Sub NotifyIcon2_MouseDown(ByVal sender As
System.Object, ByVal e As _
System.Windows.Forms.MouseEventArgs) Handles
NotifyIcon2.MouseDown

End Sub
```

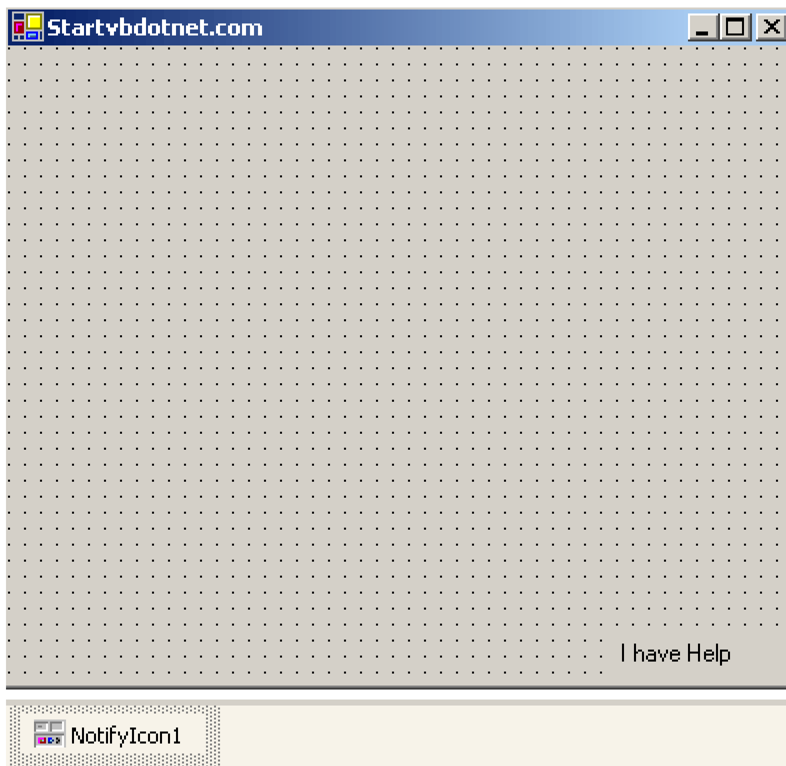
You can also handle click and double-click events for notify icon. The code sample below works with the click event of the Notify Icon to display a help file.

To create a Notify Icon component you need an icon (.ico) file to assign to it's Icon property. If you have an icon then you can use it else you might need to create an icon. You can create new icons with Visual Studio's icon designer. To open the icon designer select Project->Add New Item and from the Add New Item dialog select Icon File and click open. You can use the toolbars that are visible to design your icon. The Icon Designer Window is displayed below.



Sample Code

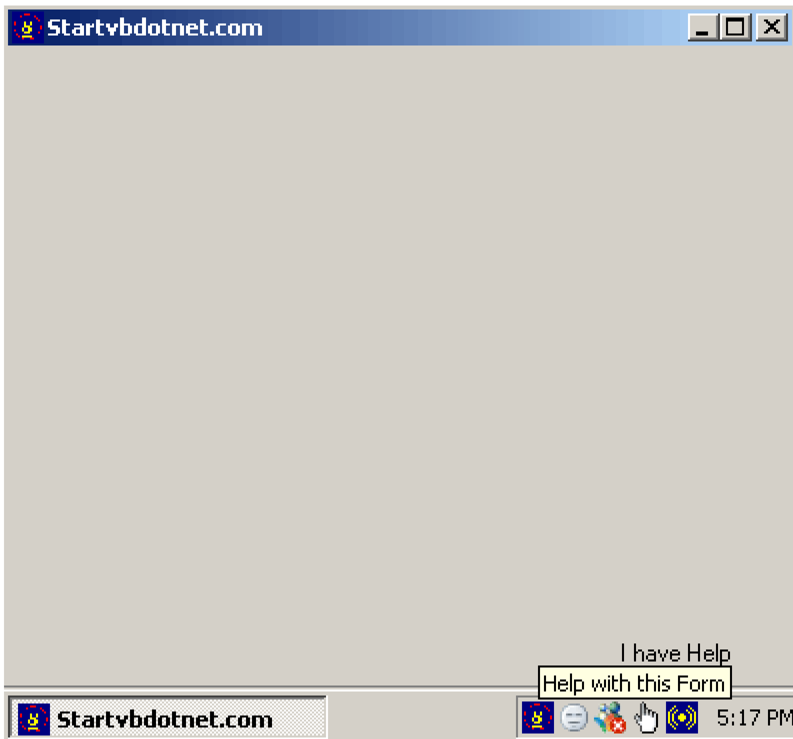
Drag a Notify Icon component and a Label control from the toolbar onto the form. Open the properties window for the Notify Icon and set the Icon property to the path of the icon and the text property to "Help with this Form". This is the icon that will be displayed when you run the application. The Label control is needed to set the help file. Set the text for label as "I have Help". The form in design view should look like the image below.



This sample code launches a help file when you click the Icon in System Tray. This sample code assumes that you have a help file, "Help.htm" in the C: drive of your machine.

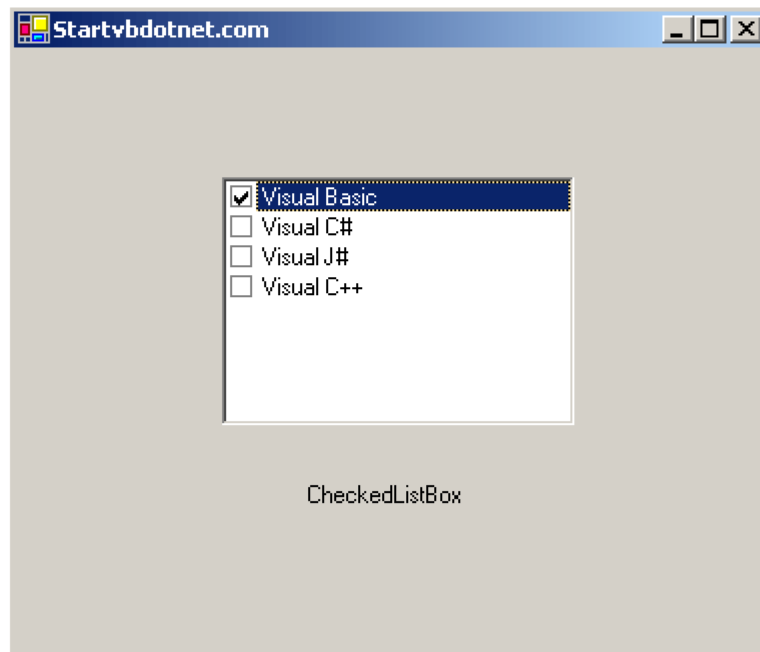
```
Private Sub NotifyIcon1_click(ByVal sender As System.Object, ByVal
e As _
System.EventArgs) Handles NotifyIcon1.Click
'handling click event of the NotifyIcon
Help.ShowHelp(Label1, "c:\help.htm")
'using the Help class and it's ShowHelp method to display a help file
End Sub
```

When you run the application, an icon will be visible in the System Tray and when you click the icon the help file named "Help.htm" will be launched. The image below displays the output from above code.



CheckedListBox

As the name says, CheckedListBox is a combination of a ListBox and a CheckBox. It displays a ListBox with a CheckBox towards it's left. The CheckedListBox class is derived from the ListBox class and is based on that class. Since the CheckedListBox is derived from the ListBox it shares all the members of ListBox. Below is the Image of a CheckedListBox.



Notable Properties of CheckedListBox

The notable property in the appearance section of the properties window is the ThreeDCheckBoxes property which is set to False by default. Setting it to True makes the CheckedListBox to be displayed in Flat or Normal style.

Notable property in the behavior section is the CheckOnClick property which is set to False by default. When set to False it means that to check or uncheck an item in the CheckedListBox we need to double-click the item. Setting it to True makes an item in the CheckedListBox to be checked or unchecked with a single click.

Notable property in the Data section is the Items property with which we add items to the CheckedListBox.

```
Private Sub CheckedListBox1_SelectedIndexChanged(ByVal sender
As System.Object,_
ByVal e As System.EventArgs) Handles
CheckedListBox1.SelectedIndexChanged

End Sub
```

Working with CheckedListBoxes is similar to working with ListBoxes.

2.1.2 Container

Building a **container control** for WinForms was easier than I first imagined. Why did I build one? Because I needed a custom solution for viewing controls that were created, but could not be placed into any other third-party control.

It appears that Microsoft left out the Data Repeater that was in VB 6 for Windows, so now we have to make our own.

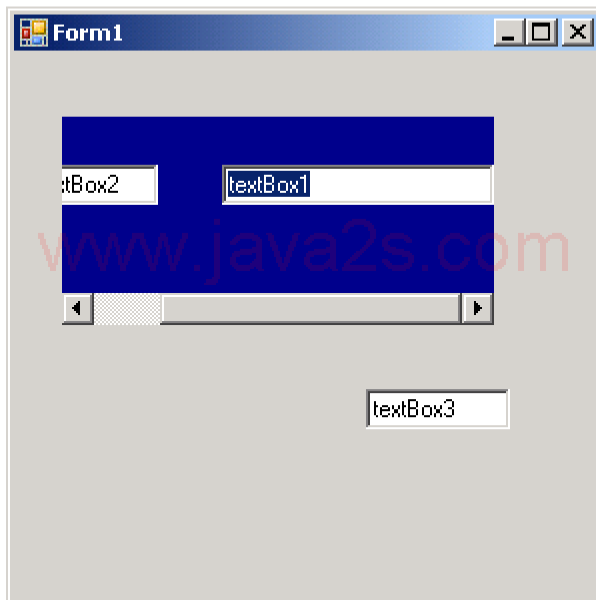
Making the Container

The container is just a UserControl. Some of the particulars are to set the AutoScroll=True and add some events for handling which control is selected as well as the count.

Reference Link For container control is below:

http://www.codeproject.com/KB/cpp/CustomContainerControl.aspx

Container Control Demo



www.java2s.com

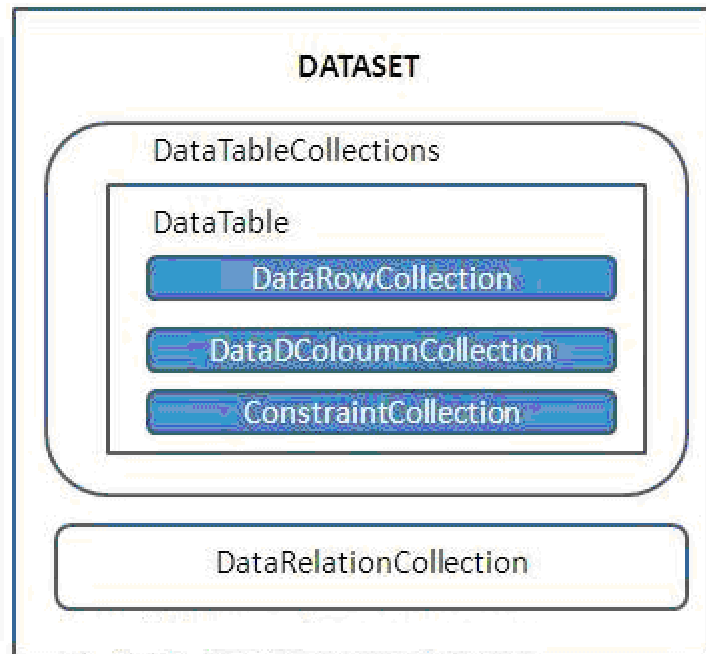
```
Imports System
Imports System.Drawing
Imports System.Windows.Forms

public class MainClass

   Shared Sub Main()
      Dim form1 As Form = New Form1
      Application.Run(form1)
   End Sub

End Class



Public Class Form1
    Inherits System.Windows.Forms.Form
```

```
#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call
    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolea
n)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
    Friend WithEvents ctl1 As System.Windows.Forms.ContainerControl
    Friend WithEvents textBox2 As System.Windows.Forms.TextBox
    Friend WithEvents textBox1 As System.Windows.Forms.TextBox
    Friend WithEvents textBox3 As System.Windows.Forms.TextBox

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the Windows Form Desi
gner
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThroughAttribute()> Private Sub Ini
tializeComponent()
        Me.textBox1 = New System.Windows.Forms.TextBox()
        Me.ctl1 = New System.Windows.Forms.ContainerControl()
        Me.textBox2 = New System.Windows.Forms.TextBox()
        Me.textBox3 = New System.Windows.Forms.TextBox()
        Me.ctl1.SuspendLayout()
        Me.SuspendLayout()
        '
        'textBox1
        '
        Me.textBox1.Location = New System.Drawing.Point(128, 24)
        Me.textBox1.Name = "textBox1"
        Me.textBox1.Size = New System.Drawing.Size(136, 20)
        Me.textBox1.TabIndex = 0
        Me.textBox1.Text = "textBox1"
        '
        'ctl1
        '
        Me.ctl1.AutoScroll = True
        Me.ctl1.BackColor = System.Drawing.Color.DarkBlue
        Me.ctl1.Controls.AddRange(New System.Windows.Forms.Control() {M
e.textBox2, Me.textBox1})
```

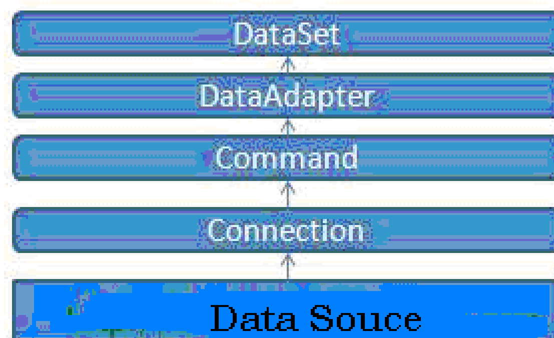
```
        Me.ctl1.Location = New System.Drawing.Point(24, 32)
        Me.ctl1.Name = "ctl1"
        Me.ctl1.Size = New System.Drawing.Size(216, 104)
        Me.ctl1.TabIndex = 0
        '
        'textBox2
        '
        Me.textBox2.Location = New System.Drawing.Point(32, 24)
        Me.textBox2.Name = "textBox2"
        Me.textBox2.Size = New System.Drawing.Size(64, 20)
        Me.textBox2.TabIndex = 1
        Me.textBox2.Text = "textBox2"
        '
        'textBox3
        '
        Me.textBox3.Location = New System.Drawing.Point(176, 168)
        Me.textBox3.Name = "textBox3"
        Me.textBox3.Size = New System.Drawing.Size(72, 20)
        Me.textBox3.TabIndex = 1
        Me.textBox3.Text = "textBox3"
        '
        'Form1
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(292, 273)
        Me.Controls.AddRange(New System.Windows.Forms.Control() {Me.tex
tBox3, Me.ctl1})
        Me.Name = "Form1"
        Me.Text = "Form1"
        Me.ctl1.ResumeLayout(False)
        Me.ResumeLayout(False)

    End Sub

#End Region

End Class
```

What is ADO.NET Dataset

The ADO.NET DataSet contains DataTableCollection and their DataRelationCollection . It represents a collection of data retrieved from the Data Source. We can use Dataset in combination with DataAdapter class. The DataSet object offers a disconnected data source architecture. The Dataset can work with the data it contain, without knowing the source of the data coming from. That is , the Dataset can work with a disconnected mode from its Data Source . It gives a better advantage over DataReader , because the DataReader is working only with the connection oriented Data Sources.



The Dataset contains the copy of the data we requested. The Dataset contains more than one Table at a time. We can set up Data Relations between these tables within the DataSet. The data set may comprise data for one or more members, corresponding to the number of rows.



The DataAdapter object allows us to populate DataTables in a DataSet. We can use Fill method of the DataAdapter for populating data in a Dataset. The DataSet can be filled either from a data source or dynamically. A DataSet can be saved to an XML file and then loaded back into memory very easily.

DataGrid

The datagrid webserver control is power tool displaying information from datasource.It is a full-featured data-bound control that displays data in tabular format, and provides the ability to sort, select, edit, and delete records from its associated data source.

ImageList

The **ImageList** component has been in Visual Basic since VB6 days. Like everything in .NET, it's changed a bit, however.

The main advantage of using the **ImageList** is that you can treat the images as a collection. The major design-time alternative is to use **Resources** instead. Doing this way, you would add all of your graphics into the **Resources** tab of the project properties. You would have to work with them individually, however, using code that looks like this:

PictureBox.Image = My.Resources.Image0

Another advantage of the **ImageList** is that the images are added into your project assembly for easy distribution and fast execution. **ImageList** also works seamlessly with other VB.NET controls such as **TreeView**, **ListView**, **TabStrip**, and **ImageCombo**. And **TabControl**. To demonstrate how that works, let's add a series of images from an **ImageList** to a **TabControl**.

Timer Control

A **Timer** control raises an event at a given interval of time without using a secondary thread. If you need to execute some code after certain interval of time continuously, you can use a timer control.

Timer Properties

Enabled property of timer represents if the timer is running. We can set this property to true to start a timer and false to stop a timer.

Interval property represents time on in milliseconds, before the Tick event is raised relative to the last occurrence of the Tick event. One second equals to 1000 milliseconds. So if you want a timer event to be fired every 5 seconds, you need to set Interval property to 5000.

```
Dim Timer1 As New Timer()
Timer1.Interval = 2000
Timer1.Enabled = True
```

Creating a Timer

A Timer control does not have a visual representation and works as a component in the background.

Design-time

You can create a timer at design-time by dragging and dropping a Timer component from Toolbox to a Form. After that, you can use F4 or right click Properties menu to set a

Timer properties as shown in Figure 1. As you can see in Figure 1, the Enabled property is false and Interval is set to 1000 milliseconds (1 second).

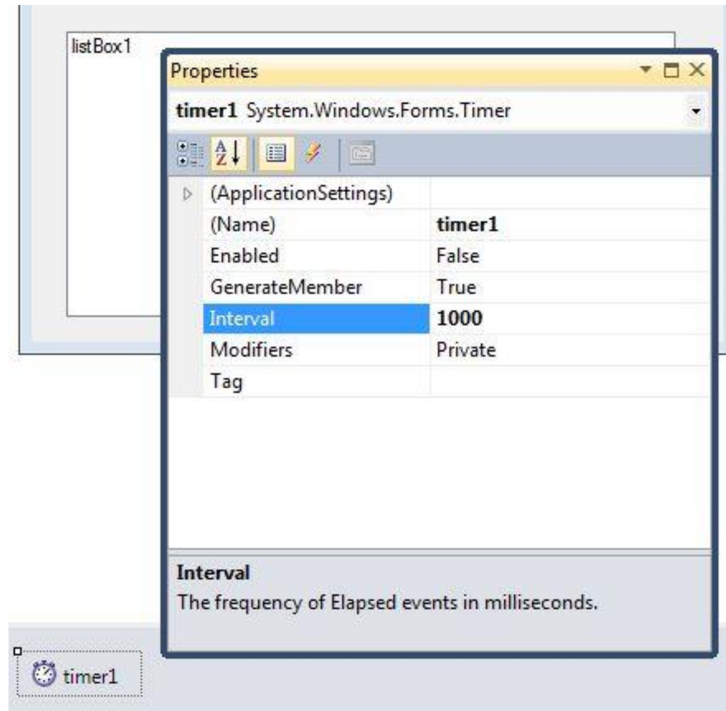


Figure 1

First thing you want to do is, change Enabled to true so the timer will start when your application starts.

Now next step is to add an event handler. If you go to the Events window by clicking little lightning icon, you will see only one Tick event as you can see from Figure 2. Double click on it will add the Tick event handler.

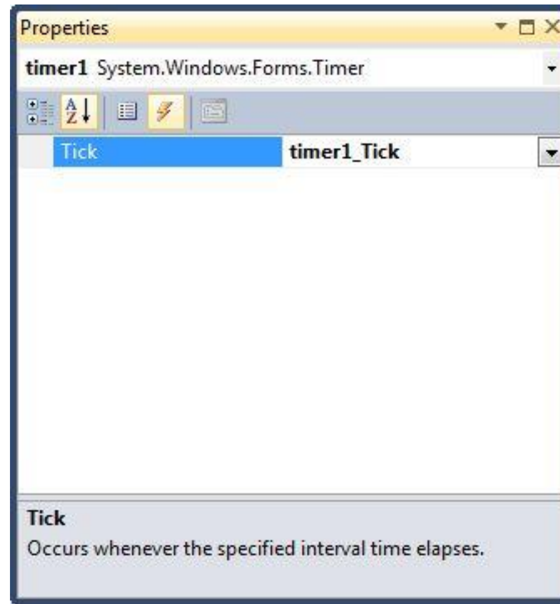


Figure 2

Now whatever code you write on this event handler, it will be executed every 1 second. For example, if you have a ListBox control on a Form and you want to add some items to it, the following code will do so.

```
Private Sub Timer1_Tick(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                        Handles Timer1.Tick
        ListBox1.Items.Add(DateTime.Now.ToLongTimeString() + "," + _
                           DateTime.Now.ToLongDateString())
End Sub
```

Run-time

Timer class represents a Timer control and used to create a Timer at run-time. The following code snippet creates a Timer at run-time, sets its property and event handler.

```
Dim t As New Timer()
t.Interval = 2000
t.Enabled = True
AddHandler t.Tick, AddressOf TimerEventHandler
```

The event handler code looks like following.

```
Private Sub TimerEventHandler(ByVal obj As Object, ByVal ergs As EventArgs)
        ListBox1.Items.Add(DateTime.Now.ToLongTimeString() + "," + _
                         DateTime.Now.ToLongDateString())
End Sub
```

Summary

In this article, we discussed discuss how to create a Timer control in Windows Forms and set its various properties and events.

Menus

Everyone should be familiar with Menus. Menus (File, Edit, Format etc in all windows applications) are those that allow us to make a selection when we want to perform some action with the application, for example, to format the text, open a new file, print and so on. In VB .NET MainMenu is the container for the Menu structure of the form. Menus are made of MenuItem objects that represent individual parts of a menu (like File->New, Open, Save, Save As etc). The two main classes involved in menu handling are, MainMenu and MenuItem. The MainMenu class let's us assign objects to a form's menu class and MenuItem is the class which supports the items in a menu system. Menus like File, Edit, Format etc and the items in those Menus are supported by this MenuItem class. It's this MenuItem's click event that makes these Menus work. For a MenuItem to be displayed, we need to add it to a MainMenu object.

Event of the MenuItem

The default event of the MenuItem is the Click event which looks like this in code:

```
Private Sub MenuItem1_Click(ByVal sender As System.Object, ByVal
e As_
System.EventArgs) Handles MenuItem1.Click

End Sub
```

Notable properties of the MenuItem class are summarized below.

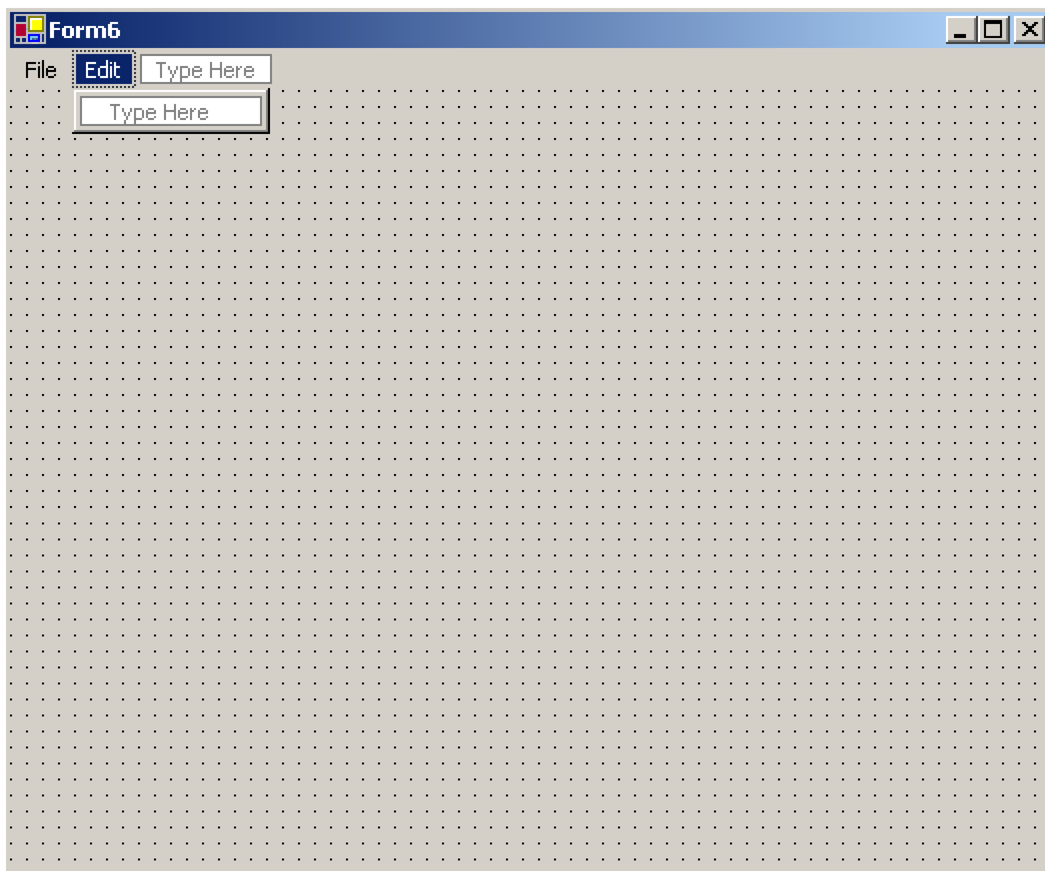
Under the Miscellaneous Section of the properties window:

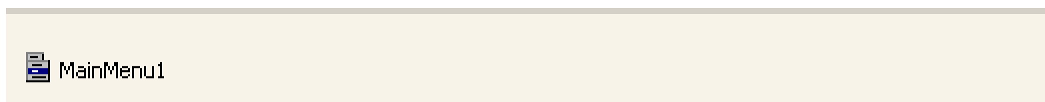
Checked: Default value is set to False. Changing it to True makes a checkmark appear towards the left of the Menu.
DefaultItem: Default value is set to False. Changing it to True makes this menu item default menu item.
RadioCheck: Changing it to True makes a menu item display a radio button instead of a checkmark.
Shortcut: Enables to set a short cut key from a list of available shortcuts for the menu item.

Working with Menus

Creating Menus is simple. Drag a MainMenu component from the toolbar onto the form. When you add a MaiuMenu component to the form it appears in the component tray below the form. Windows form designer will add the MenuItem's for this by default, you need not add this. Once when you finish adding a MainMenu component to the form you will notice a "TypeHere" box towards the top-left corner of the form. To create a menu all you have to do is click on the "TypeHere" text which opens up a small textbox allowing you to enter text for the menu. You can view that in the image below. You can

use the arrow keys on the keyboard to create a submenu or add other items to that menu or click on the first menu item and use the left/right arrow keys on the keyboard to create a new menu item. That's all it takes to add a menu to the form.





Working with an example

Let's work with an example to understand Menus. Drag a MainMenu and a TextBox onto the form. In the "Type Here" part, type File and under file type "New" and "Exit". Our intention here is to display "Welcome to Menus" in the TextBox when "New" is clicked and close the form when "Exit" is clicked. The Menu which we will create should look like this File->New, Exit (New and Exit below File). The code for that looks like this:

```
Public Class Form3 Inherits System.Windows.Forms.Form
#Region " Windows Form Designer generated code "

Private Sub MenuItem2_Click(ByVal sender As System.Object, ByVal
e_
As System.EventArgs)_ Handles MenuItem2.Click
```

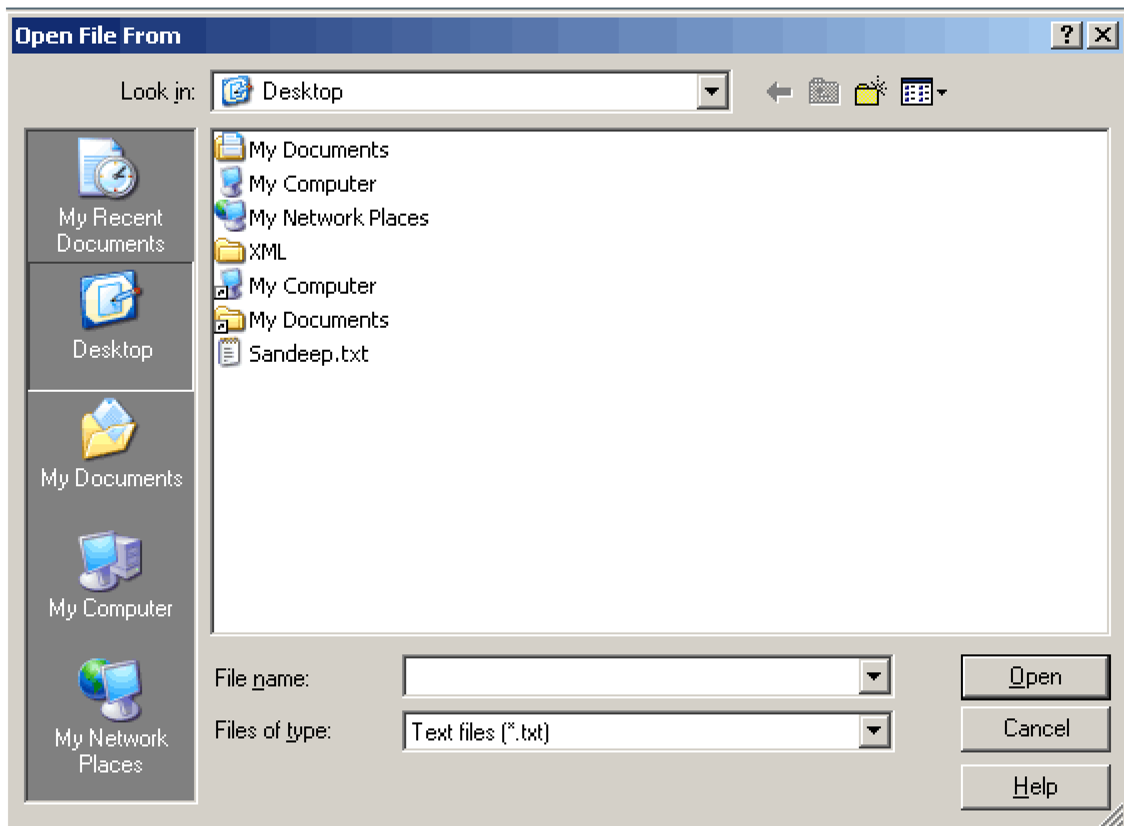
```
TextBox1.Text = "Welcome to Menus"
End Sub

Private Sub MenuItem3_Click(ByVal sender As System.Object, ByVal
e As System.EventArgs)_ Handles MenuItem3.Click
Me.Close()
'Me refers to the current object (form)
End Sub

End Class
```

Common Dialogs

Visual Basic .NET comes with built-in dialog boxes which allow us to create our own File Open, File Save, Font, Color dialogs much like what we see in all other windows applications. To make a dialog box visible at run time we use the dialog box's ShowDialog method. The Dialog Boxes which come with Visual Basic .NET are: OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog, PrintDialog, PrintPreviewDialog and PageSetupDialog. We will be working with OpenFile, SaveFile, Font and Color Dialog's in this section. The return values of all the above said dialog boxes which will determine which selection a user makes are: Abort, Cancel, Ignore, No, None, OK, Return, Retry and Yes.

OpenFileDialog

Open File Dialog's are supported by the OpenFileDialog class and they allow us to select a file to be opened. Below is the image of an OpenFileDialog.



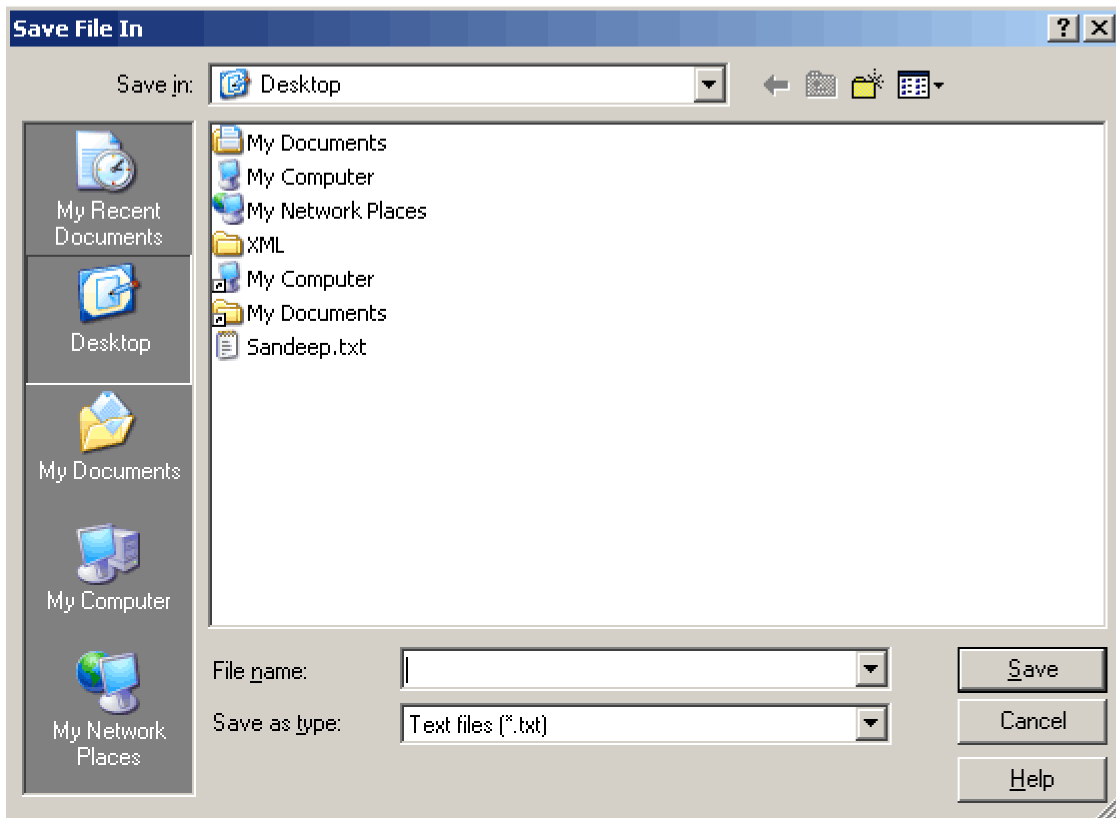
Properties of the OpenFileDialog are as follows:

AddExtension: Gets/Sets if the dialog box adds extension to file names if the user doesn't supply the extension.
CheckFileEixsts: Checks whether the specified file exists before returning from the dialog.
CheckPathExists: Checks whether the specified path exists before returning from the dialog.
DefaultExt: Allows you to set the default file extension.
FileName: Gets/Sets file name selected in the file dialog box.
FileNames: Gets the file names of all selected files.
Filter: Gets/Sets the current file name filter string, which sets the choices that appear in the "Files of Type" box.
FilterIndex: Gets/Sets the index of the filter selected in the file dialog box.
InitialDirectory: This property allows to set the initial directory which should open when you use the OpenFileDialog.
MultiSelect: This property when set to True allows to select multiple file extensions.
ReadOnlyChecked: Gets/Sets whether the read-only checkbox is checked.
RestoreDirectory: If True, this property restores the original directory before closing.
ShowHelp: Gets/Sets whether the help button should be displayed.
ShowReadOnly: Gets/Sets whether the dialog displays a read-only check box.
Title: This property allows to set a title for the file dialog box.

ValidateNames: This property is used to specify whether the dialog box accepts only valid file names.

SaveFileDialog

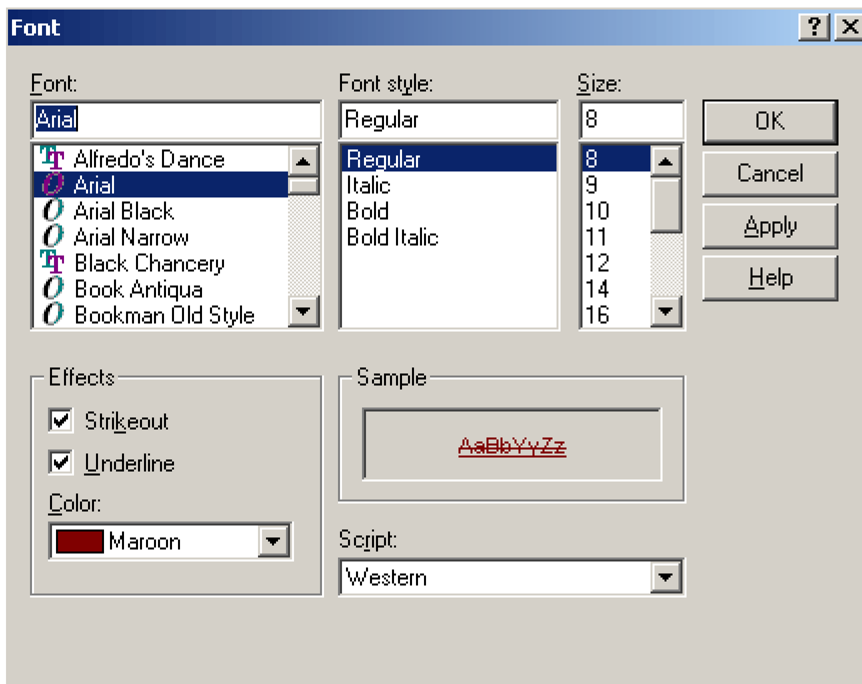
Save File Dialog's are supported by the SaveFileDialog class and they allow us to save the file in a specified location. Below is the image of a SaveFileDialog.



Properties of the Save File Dialog are the same as that of the Open File Dialog. Please refer above. Notable property of Save File dialog is the OverwritePromopt property which displays a warning if we choose to save to a name that already exists.

FontDialog

Font Dialog's are supported by the FontDialog Class and they allow us to select a font size, face, style, etc. Below is the image of a FontDialog.

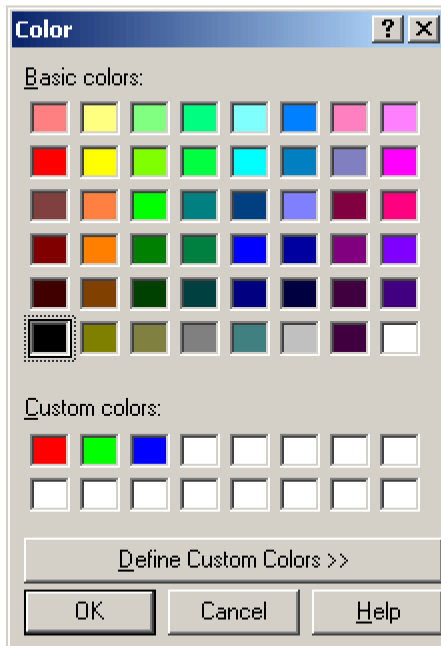


Properties of the FontDialog are as follows:

AllowSimulations: Gets/Sets whether the dialog box allows graphics device interface font simulations.
AllowVectorFonts: Gets/Sets whether the dialog box allows vector fonts.
AllowVerticalFonts: Gets/Sets whether the dialog box displays both vertical and horizontal fonts or only horizontal fonts.
Color: Gets/Sets selected font color.
FixedPitchOnly: Gets/Sets whether the dialog box allows only the selection of fixed-pitch fonts.
Font: Gets/Sets the selected font.
FontMustExist: Gets/Sets whether the dialog box specifies an error condition if the user attempts to select a font or size that doesn't exist.
MaxSize: Gets/Sets the maximum point size the user can select.
MinSize: Gets/Sets the mainimum point size the user can select.
ShowApply: Gets/Sets whether the dialog box contains an apply button.
ShowColors: Gets/Sets whether the dialog box displays the color choice.
ShowEffects: Gets/Sets whether the dialog box contains controls that allow the user to specify to specify strikethrough, underline and text color options.
ShowHelp: Gets/Sets whether the dialog box displays a help button.

ColorDialogs

Color Dialog's are supported by the ColorDialog Class and they allow us to select a color. The image below displays a color dialog.



Properties of ColorDialog are as follows:

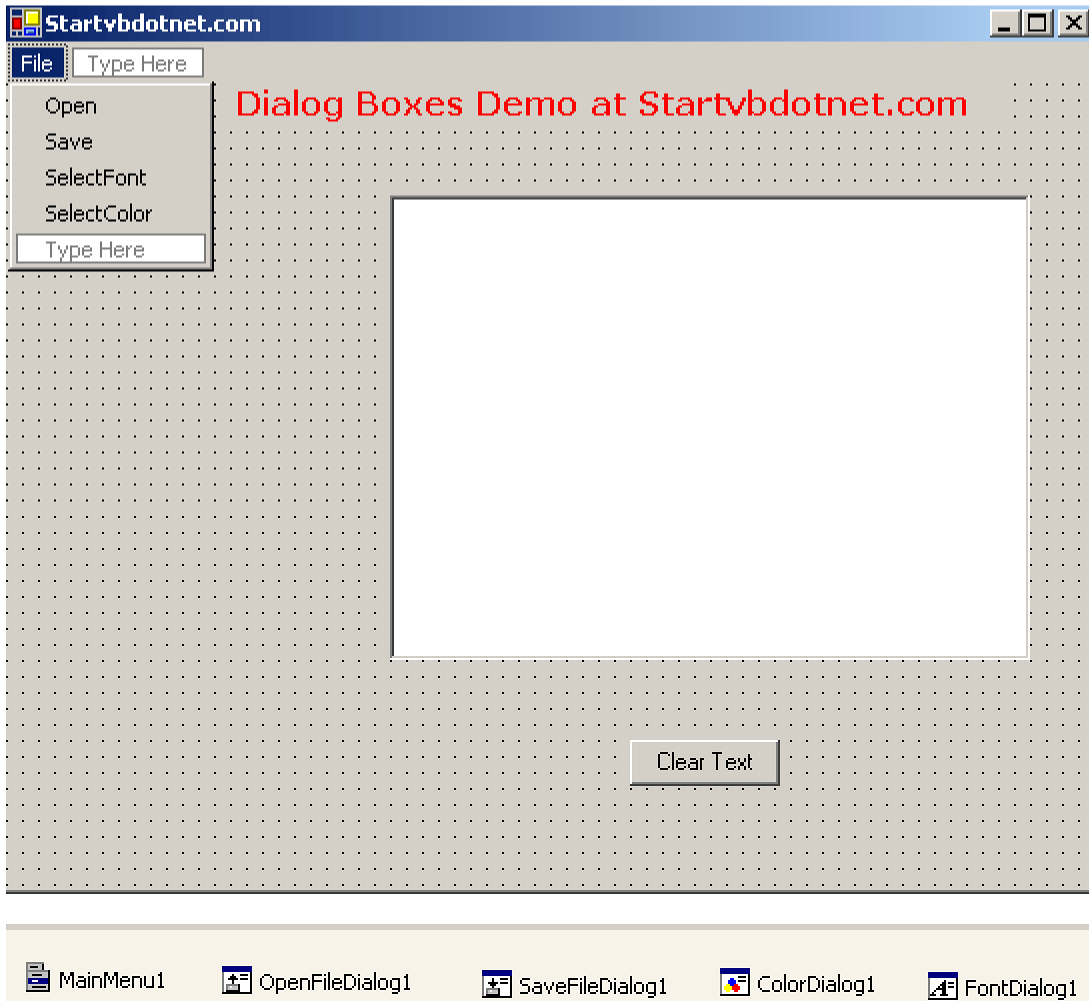
AllowFullOpen: Gets/Sets whether the user can use the dialog box to define custom colors.
AnyColor: Gets/Sets whether thedialog box displays all the available colors in the set of basic colons.
Color: Gets/Sets the color selected by the user.
CustomColors: Gets/Sets the set of custom colors shown in the dialog box.
FullOpen: Gets/Sets whether the controls used to create custom colors are visible when the dialog box is opened.
ShowHelp: Gets/Sets whether the dialog box displays a help button.
SolidColorOnly: Gets/Sets whether the dialog box will restrict users to selecting solid colors only.

Dialog Boxes

Putting Dialog Boxes to Work

We will work with OpenFile, SaveFile, Font and Color Dialog's in this section. From the toolbox drag a MainMenu component, RichTextBox control, Button Control, OpenFileDialog, SaveFileDialog, FontDialog and ColorDialog onto the form. The sample code demonstrated below allows you to select a file to be opened and displays it in the RichTextBox with OpenFileDialog, allows you to save the text you enter in the RichTextBox to a location using the SaveFileDialog, allows you to select a font and applies the selected font to text in the RTB using FontDialog and allows you to select a color and applies the color to text in the RTB using the ColorDialog. Select the MainMenu component and in the "Type Here" part of the MainMenu type File and using

the down arrow keys on the keyboard start typing Open, Save, SelectFont and SelectColor under the File menu. It should look like this: File-> Open, Save, SelectFont, SelectColor. We will assign OpenFileDialog to Open, SaveFileDialog to Save, FontDialog to SelectFont and ColorDialog to SelectColor under File Menu. The form in design view should look similar to the image below.



Before proceeding further you need to set properties for these dialogs in their properties window. They are listed below.

For OpenFileDialog1, set the DefaultExt property to txt so that it opens text files, InitialDirectory property to C:, RestoreDirectory propery to True and the Text property to Open File From.

For SaveFileDialog1, set the DefaultExt property to txt so that it saves files in text format, InitialDirectory property to C: so that when you save a file, it first provides C: drive as the choice of location, OverwritePrompt property to False, RestoreDirectory propery to True and the Text property to Save File In.

For FontDialog1, set the AllowSimulations, AllowVectorFonts, AllowverticalFonts properties to false, MaxSize to 50, MinSize to 5 and ShowApply and ShowColor properties to True.

For ColorDialog1, set AnyColor and SolidColorOnly properties to True.

Code

```
Imports System.IO
Public Class Form1 Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

#End Region

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As_
System.EventArgs) Handles Button1.Click
RichTextBox1.Text = " "
'clears the text in richtextbox
End Sub

Private FileName As String
'declaring filename that will be selected
Dim sr As StreamReader
'streamreader is used to read text

Private Sub MenuItem2_Click(ByVal sender As System.Object, ByVal
e As_
System.EventArgs) Handles MenuItem2.Click
Try
With OpenFileDialog1
'With statement is used to execute statements using a particular object,
here,_
'OpenFileDialog1
.Filter = "Text files (*.txt)|*.txt|" & "All files|*.*"
'setting filters so that Text files and All Files choice appears in the Files
of Type box
'in the dialog
If .ShowDialog() = DialogResult.OK Then
'showDialog method makes the dialog box visible at run time
FileName = .FileName
sr = New StreamReader(.OpenFile)
'using streamreader to read the opened text file
RichTextBox1.Text = sr.ReadToEnd()
'displaying text from streamreader in richtextbox
```

```
End If
End With
Catch es As Exception
MessageBox.Show(es.Message)
Finally
If Not (sr Is Nothing) Then
sr.Close()
End If
End Try
End Sub

Private Sub MenuItem3_Click(ByVal sender As System.Object, ByVal
e As_
System.EventArgs) Handles MenuItem3.Click
Dim sw As StreamWriter
'streamwriter is used to write text
Try
With SaveFileDialog1
.FileName = FileName
.Filter = "Text files (*.txt)|*.txt|" & "All files|*.*"
If .ShowDialog() = DialogResult.OK Then
FileName = .FileName
sw = New StreamWriter(FileName)
'using streamwriter to write text from richtextbox and saving it
sw.Write(RichTextBox1.Text)
End If
End With
Catch es As Exception
MessageBox.Show(es.Message)
Finally
If Not (sw Is Nothing) Then
sw.Close()
End If
End Try
End Sub

Private Sub MenuItem4_Click(ByVal sender As System.Object, ByVal
e As_
System.EventArgs) Handles MenuItem4.Click
Try
With FontDialog1
.Font = RichTextBox1.Font
'initializing the dialog box to match the font used in the richtextbox
.Color = RichTextBox1.ForeColor
'default color is Black
If .ShowDialog = DialogResult.OK Then
```

```
setFont()
'calling a method setFont() to set the selected font and color
End If
End With
Catch es As Exception
MessageBox.Show(es.Message)
End Try
End Sub

Private Sub setFont()
Try
With FontDialog1
RichTextBox1.Font = .Font
If .ShowColor Then
RichTextBox1.ForeColor = .Color
'setting the color
End If
End With
Catch ex As Exception
MessageBox.Show(ex.Message)
End Try
End Sub

Private Sub MenuItem5_Click(ByVal sender As System.Object, ByVal
e As _
System.EventArgs) Handles MenuItem5.Click
Static CustomColors() As Integer = {RGB(255, 0, 0), RGB(0, 255, 0),
RGB(0, 0, 255)}
'initializing CustomColors with an array of integers and putting Red,
Green,
'and Blue in the custom colors section
Try
With ColorDialog1
.Color = RichTextBox1.ForeColor
'initializing the selected color to match the color currently used
'by the richtextbox's foreground color
.CustomColors = CustomColors
'filling custom colors on the dialog box with the array declared above
If .ShowDialog() = DialogResult.OK Then
RichTextBox1.ForeColor = .Color
CustomColors = .CustomColors
'Storing the custom colors to use again
End If
ColorDialog1.Reset()
'resetting all colors in the dialog box
End With
```

```
Catch es As Exception
MessageBox.Show(es.Message)
End Try
End Sub

End Class
```

2.3 Exception Handling

Exception handling is an in built mechanism in .NET framework to detect and handle run time errors. The .NET framework contains lots of standard exceptions. The exceptions are anomalies that occur during the execution of a program. They can be because of user, logic or system errors. If a user (programmer) do not provide a mechanism to handle these anomalies, the .NET run time environment provide a default mechanism, which terminates the program execution.

VB.NET provides three keywords try, catch and finally to do exception handling. The try encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for doing any clean up process.

The general form try-catch-finally in VB.NET is shown below.

```
Try
' Statement which can cause an exception.
Catch x As Type
' Statements for handling the exception
Finally
End Try 'Any cleanup code
```

If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.

But in VB.NET, both catch and finally blocks are optional. The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

If there is no exception occurred inside the try block, the control directly transfers to finally block. We can say that the statements inside the finally block is executed always. Note that it is an error to transfer control out of a finally block by using break, continue, return or goto.

In VB.NET, exceptions are nothing but objects of the type Exception. The Exception is the ultimate base class for any exceptions in VB.NET. The VB.NET itself provides couple of standard exceptions. Or even the user can create their own exception classes, provided that this should inherit from either Exception class or one of the standard derived classes of Exception class like DivideByZeroExcpetion ot ArgumentException etc.

Uncaught Exceptions

The following program will compile but will show an error during execution. The division by zero is a runtime anomaly and program terminates with an error

message. Any uncaught exceptions in the current context propagate to a higher context and looks for an appropriate catch block to handle it. If it can't find any suitable catch blocks, the default mechanism of the .NET runtime will terminate the execution of the entire program.

```
//VB.NET: Exception Handling
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 100 / x
Console.WriteLine(div)
End Sub 'Main
End Class 'MyClient
```

The modified form of the above program with exception handling mechanism is as follows. Here we are using the object of the standard exception class DivideByZeroException to handle the exception caused by division by zero.

```
//VB.NET: Exception Handling
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 0
Try
div = 100 / x
Console.WriteLine("This line in not executed")
Catch de As DivideByZeroException
onsole.WriteLine("Exception occured")
End Try
Console.WriteLine("Result is {0}", div)
End Sub 'Main
End Class 'MyClient
```

In the above case the program do not terminate unexpectedly. Instead the program control passes from the point where exception occurred inside the try block to the catch blocks. If it finds any suitable catch block, executes the statements inside that catch and continues with the normal execution of the program statements.

If a finally block is present, the code inside the finally block will get also be executed.

```
//VB.NET: Exception Handling
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 0
Try
div = 100 / x
Console.WriteLine("Not executed line")
Catch de As DivideByZeroException
Console.WriteLine("Exception occured")
```

```
Finally
Console.WriteLine("Finally Block")
End Try
Console.WriteLine("Result is {0}", div)
End Sub 'Main
End Class 'MyClient
```

Remember that in VB.NET, the catch block is optional. The following program is perfectly legal in VB.NET.

```
//VB.NET: Exception Handling
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 0
Try
div = 100 / x
Console.WriteLine("Not executed line")
Finally
Console.WriteLine("Finally Block")
End Try
Console.WriteLine("Result is {0}", div)
End Sub 'Main
End Class 'MyClient
```

But in this case, since there is no exception handling catch block, the execution will get terminated. But before the termination of the program statements inside the finally block will get executed. In VB.NET, a try block must be followed by either a catch or finally block.

Multiple Catch Blocks

A try block can throw multiple exceptions, which can handle by using multiple catch blocks. Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

```
//VB.NET: Exception Handling: Multiple catch
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 0
Try
div = 100 / x
Console.WriteLine("Not executed line")
Catch de As DivideByZeroException
Console.WriteLine("DivideByZeroException")
Catch ee As Exception
Console.WriteLine("Exception")
Finally
Console.WriteLine("Finally Block")
End Try
```

```
Console.WriteLine("Result is {0}", div)
End Sub 'Main
End Class 'MyClient
```

Catching all Exceptions

By providing a catch block without a brackets or arguments, we can catch all exceptions occurred inside a try block. Even we can use a catch block with an Exception type parameter to catch all exceptions happened inside the try block since in VB.NET, all exceptions are directly or indirectly inherited from the Exception class.

```
//VB.NET: Exception Handling: Handling all exceptions
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 0
Try
div = 100 / x
Console.WriteLine("Not executed line")
Catch
End Try
Console.WriteLine("Result is {0}", div)
End Sub 'Main
End Class 'MyClient
```

The following program handles all exception with Exception object.

```
//VB.NET: Exception Handling: Handling all exceptions
Imports System
Class MyClient
Public Shared Sub Main()
Dim x As Integer = 0
Dim div As Integer = 0
Try
div = 100 / x
Console.WriteLine("Not executed line")
Catch e As Exception
Console.WriteLine("oException")
End Try
Console.WriteLine("Result is {0}", div)
End Sub 'Main
End Class 'MyClient
```

Throwing an Exception

In VB.NET, it is possible to throw an exception programmatically. The 'throw' keyword is used for this purpose. The general form of throwing an exception is as follows.

```
Throw exception_obj
```

For example the following statement throw an ArgumentException explicitly.

```
Throw New ArgumentException("Exception")

//VB.NET: Exception Handling:
Imports System
Class MyClient
Public Shared Sub Main()
Try
Throw New DivideByZeroException("Invalid Division")
Catch e As DivideByZeroException
Console.WriteLine("Exception")
End Try
Console.WriteLine("LAST STATEMENT")
End Sub 'Main
End Class 'MyClient
```

Re-throwing an Exception

The exceptions, which we caught inside a catch block, can re-throw to a higher context by using the keyword throw inside the catch block. The following program shows how to do this.

```
//VB.NET: Exception Handling: Handling all exceptions
Imports System
Class [MyClass]
Public Sub Method()
Try
Dim x As Integer = 0
Dim sum As Integer = 100 / x
Catch e As DivideByZeroException
Throw
End Try
End Sub 'Method
End Class '[MyClass]
Class MyClient
Public Shared Sub Main()
Dim mc As New [MyClass]
Try
mc.Method()
Catch e As Exception
Console.WriteLine("Exception caught here")
End Try
Console.WriteLine("LAST STATEMENT")
End Sub 'Main
End Class 'MyClient
```

Standard Exceptions

There are two types of exceptions: exceptions generated by an executing program and exceptions generated by the common language runtime. System.Exception is the base class for all exceptions in VB.NET. Several exception classes inherit from this class including ApplicationException and SystemException. These two classes form the basis for most other runtime exceptions. Other exceptions that derive

directly from System.Exception include IOException, WebException etc.

The common language runtime throws SystemException. A user program rather than the runtime throws the ApplicationException. The SystemException includes the ExecutionEngineException, StaclOverFlowException etc. It is not recommended that we catch SystemExceptions nor is it good programming practice to throw SystemExceptions in our applications.

System.OutOfMemoryException
System.NullReferenceException
Syste.InvalidCastException
Syste.ArrayTypeMismatchException
System.IndexOutOfRangeException
System.ArithmeticException
System.DevideByZeroException
System.OverFlowException

User-defined Exceptions

In VB.NET, it is possible to create our own exception class. But Exception must be the ultimate base class for all exceptions in VB.NET. So the user-defined exception classes must inherit from either Exception class or one of its standard derived classes.

```
//VB.NET: Exception Handling: User defined exceptions
Imports System
Class MyException
Inherits Exception
Public Sub New(ByVal str As String)
Console.WriteLine("User defined exception")
End Sub 'New
End Class 'MyException
Class MyClient
Public Shared Sub Main()
Try
Throw New MyException("RAJESH")
Catch e As Exception
Console.WriteLine(("Exception caught here" + e.ToString()))
End Try
Console.WriteLine("LAST STATEMENT")
End Sub 'Main
End Class 'MyClient
```

Structured Exception Handling

VB.NET utilizes the .NET Framework's standard mechanism for error reporting, called Structured Exception Handling; it relies on exceptions to report errors that arise in applications. Exceptions are classes that trap the error information. To utilize .NET's

Structured Exception Handling mechanisms properly, developers need to write smart code that watches out for exceptions and implement code to deal with these exceptions.

- Post a comment
- Email Article
- Print Article
- Share Articles

Structured exception handling provides the following components in the code:

- **Try section:** The block of code that may result in an exception and always gets executed
- **Catch section:** The block of code that attempts to act on an exception and is only executed when an exception takes place
- **Finally section:** The block of code intended to perform any kind of clean up operation and always gets executed

The Exception Class

Each exception class in .NET is derived from a *System.Exception* class. The most often used members of the Exception class are listed below:

- **Message:** Specifies details of an error
- **Source:** Name of the object or application that caused the exception
- **TargetSite:** Name of the method that threw the exception

The Try...Catch Block

The purpose of the *Try...Catch* block is to allow catching errors and specifying a resolution for them. The sample code looks like this:

```
Try
   'Code to be executed
Catch
   'Error resolution code
End Catch
```

Use the *Try* section to write the code that should be executed and the *Catch* section to catch and act on any errors that may have been generated while executing the code in the *Try* section. The protected code appearing in the *Try* section always gets executed; however, the code in the *Catch* section is executed only if an error occurs. The *Try* section of the code always gets executed; however, the *Catch* section of the code will be executed only if an error occurred.

The Try...Catch...Finally Block

The purpose of the *Try...Catch...Finally* block is to allow executing the protected code under the Try section, acting on any errors that may arise in the *Catch* block, and following up with the cleanup code in the *Finally* block. Code under the *Finally* block will be executed regardless of whether an error occurred in the *Try* code block or not. This provides a very convenient way of ensuring that allocated resources are being cleaned and performing any kind of functionality that needs to take place regardless of the error handling details. The sample code looks like this:

```
Try
   'Code to be executed
Catch
   'Error resolution code
Finally
   'Cleanup code
End Catch
```

The *Try* and *Finally* sections of the code always get executed. However, the Catch section of the code will be executed only if an error occurred.

Catching All Exceptions vs. Specific Classes of Exceptions

The structured exception handling in .NET is flexible and allows catching a specific type of an exception or any exception, depending on how you utilize it.

Example: Catching any exception that may occur

```
Try
   Dim i As Integer = 0
   Dim iresult As Integer

   iresult = 1 / i

Catch ex As Exception
   MessageBox.Show(ex.ToString())

Finally
   MessageBox.Show("finally block executed")
End Try
```

How this works

In the code example above, you purposefully create a run-time error to demonstrate catching any exception. You catch any error and respond to it regardless of what kind of error occurred. The error takes place in the *Try* code block, so when the exception is raised it follows to the *Catch* code block and then to the *Finally* code block. You catch the exception by declaring a variable, *ex*, of the *Exception* type.

Example: Catching a specific Exception

```
Try
```

```
   Dim i As Integer = 0
   Dim iresult As Integer

   iresult = 1 / i

Catch ex As OverflowException
   MessageBox.Show(ex.ToString())

Finally
   MessageBox.Show("finally block executed")
End Try
```

Unstructured Error Handling

Unstructured error handling is implemented with the **On Error** statement, which is placed at the beginning of a code block to handle all possible exceptions that occur during the execution of the code. All Visual Basic 6.0 error handlers in .NET are objects that can be accessed by using the **Microsoft.VisualBasic.Information.Err** namespace. The handler is set to **Nothing** each time the procedure is called. You should place only one **On Error** statement in each procedure, because additional statements disable all previous handlers that are defined in that procedure.

On Error Statement

The **On Error** statement is used to enable an error-handling routine, disable an error handling routine, or specify where to branch the code in the event of an error.

```
   On Error { GoTo [ line | 0 | -1 ] | Resume Next }
```

GoTo *line*

Used to enable the error-handling routine, starting at the location that is specified by the ***line*** argument. The ***line*** argument can be either a line label or a line number that is located within the closing procedure. A run-time error activates the error handler and branches the control to the specified line. If the specified line is not located within the same procedure as the **On Error** statement, a compile error occurs.

To avoid unexpected behavior, place an **Exit Sub** statement, an **Exit Function** statement, or an **Exit Property** statement just before the line label or line number. This prevents the error-handling code from running when no error has occurred.

GoTo 0

Disables the enabled error handler that is defined within the current procedure and resets it to **Nothing**.

GoTo -1

Disables the enabled exception that is defined within the current procedure and resets it to **Nothing**.

Resume Next

Moves the control of execution to the statement that follows immediately after the statement that caused the run-time error to occur, and continues the execution from this point forward. This is the preferred form to use to access objects, rather than using the **On Error GoTo** statement.

Example

In the following example code, the error handler is enabled on the first line of the routine with the **On Error GoTo Unstructured** statement. The location of the error handling routine is identified with the **Unstructured** line label. The error routine implements a simple **Select Case** statement that executes the corresponding block of code, depending on the error that occurred.

The **Resume Next** statement at the end of the error handling procedure returns control of the execution back to the line that follows the line that caused the error to occur.

The error handler is then disabled with the **On Error GoTo 0** statement, followed by the **On Error Resume Next** statement, which reactivates the error handler. If a run-time error occurs, the statement causes the execution to branch to the line that immediately follows the line that caused the error to occur, the same way that the **Resume Next** statement does in the error handling routine. In this case, that line is the **If** statement that evaluates the error number and displays it to the user, as well as clearing the error object.

```
   Public Sub fnErrors()
   On Error GoTo Unstructured ' Enable error handler

      Dim Result As Integer
      Dim Value1 As Integer = 9
      Dim Value2 As Integer = 0

      On Error GoTo 0 ' Disables the error handler

      'Moves execution to the line following the line that caused the
error.
      On Error Resume Next

      Result = Value1 / Value2 ' Division by zero, cause an overflow
error.

      ' Catch the overflow error caused by dividing by zero.
      If Err.Number = 6 Then
        MessageBox.Show("Error Number: " & Err.Number.ToString)
        Err.Clear() ' Clear Errors
      End If
      Exit Sub

   Unstructured:  ' Location of error handler
      Select Case Err.Number
        Case 6
```

```
        ' Display the error number.
        MessageBox.Show("Divided by zero")
      Case Else
        ' Catch all other type of errors.
        MessageBox.Show(Err.Description)
    End Select
    'Resume execution to the line following the line that caused the
error.
    Resume Next
  End Sub
```

Arrays and Collections

The .NET Framework provides many different collection classes, from the generic **Array** to the very specialized **NameValueCollection**. These classes include:

Array is a fixed-capacity construct that can have one or more dimensions. The .NET Framework supports multi-dimensional arrays similar to Pascal arrays, and jagged arrays such as you find in C.

ArrayList is a one-dimensional, expandable list that provides some advanced capabilities like the ability to add, insert, or remove multiple items in a single operation.

Hashtable is a collection in which each entry consists of a name/value pair. **Hashtable** allows for quick retrieval of an item based on its hash key.

SortedList is something of a cross between a **Hashtable** and an **ArrayList**. The items in a **SortedList** can be accessed by index, by key, or by value.

Queue and **Stack** implement the common queue and stack data structures. A **Queue** is a first-in-first-out construct that implements three primary operations: **Enqueue**, **Dequeue**, and **Peek**. **Stack**, like the stack in a processor, implements **Push**, **Pop**, and **Peek**. These two collections are very useful for temporary storage, and for implementing common algorithms that use those data structures.

Bit collections provide efficient storage of and access to collections of single-bit flags. Methods allow you to apply Boolean filters to ranges of flags.

In addition to these and other built-in collection types, the .NET Framework provides base classes (**CollectionBase** and **NameValueCollectionBase**, for example) that allow you to define your own generic or strongly-typed collections.

An **Array** is a .NET Framework object that implements the **IList** interface. Arrays in .NET can have any number of dimensions, and can be rectangular like arrays in Pascal and Basic, or jagged as in C and C++. However, jagged arrays are not compliant with the Common Language Specification, so if you're writing CLS-compliant code you can't use jagged arrays as method parameters or return types.

The .NET Framework does not require that an array's lower bound be zero. However, the C# and Visual Basic language definitions *do* impose this restriction. In C# and Visual Basic, the lower bound of an **Array** is zero. Other .NET languages (Pascal, for example) do not have this restriction.

The **Array** class is abstract, but user programs are not allowed to create subclasses of **Array**. Only the system and compilers are allowed to inherit directly from the **Array** class. To create an array in your language, you use the language-defined array construct. For example, the code below shows how to create, initialize, and access an array in C# and in Visual Basic.

C#

```
static void Main(string[] args)
{
  int[] a = new int[] {1, 2, 3, 4, 5};
  for (int i = 0; i < a.Length; i++)
  {
    Console.WriteLine(a[i]);
  }
}
```

Visual Basic

```
Sub DoArray()
  Dim a() As Integer = {1, 2, 3, 4, 5}
  Dim i As Integer
  For i = 0 To a.Length - 1
    Console.WriteLine(a(i))
  Next
End Sub
```

There are several different ways to create and initialize an array, depending on the language you're using. The example above declares an open-ended array and then initializes it with five values. This results in an array with a single row of five items

As I pointed out, arrays in Visual Basic and C# have a lower bound of zero. An array with five items, then, has index values 0 through 4. An attempt to access an item outside the bounds of the array will result in an **IndexOutOfRangeException**.

If you want to declare an array with a specific number of items, but not initialize it, you place the number of elements in the parentheses (Visual Basic) or braces (C#), like this:

C#

```
int[] a = new int[5];
```

Visual Basic

```
Dim a as Int(5)
```

In C#, you can also write:

```
int[] a = new int[5] {1, 2, 3, 4, 5};
```

In C#, the number you supply in brackets is the number of items in the array. In this case, you are creating an array that has five items, numbered 0 through four. In Visual Basic, however, the number in parentheses specifies the *upper bound* of the array. So the array defined above actually holds 6 items, numbered 0 through 5. Be very careful about this distinction when you're trying to translate examples between Visual Basic and C#.

You can determine how many elements are in a one-dimensional array by querying the array's **Length** property, as shown in the example code. Again, **Length** returns the number of items in the array. In a one-dimensional array, the upper bound is always one less than the length.

Multi-dimensional Arrays

Multi-dimensional arrays come in two flavors: jagged arrays, as found in C and C++, and rectangular arrays, as found in Pascal and Visual Basic. The .NET Framework supports both types, but only rectangular arrays are CLS compliant. If you're writing code that must be used by other .NET languages, any multi-dimensional arrays in the public or protected interfaces of your classes must be of the rectangular variety. You will find, as you get more familiar with .NET programming, that although multi-dimensional arrays are useful, it's rarely necessary to use them in your class interfaces. The other collection types are much more flexible and usually provide the functionality that you need.

You create, initialize, and access a rectangular array in much the same way as you do a one-dimensional array, as shown here.

C#

```
static void DoRectangularArray()
{
  int[,] a = new int[3,3] {{1,2,3},{4,5,6,},{7,8,9},{10,11,12}};
  Console.WriteLine("Rank = {0}", a.Rank);
  Console.WriteLine("Length = {0}", a.Length);
  Console.WriteLine("Upper bound(0) = {0}", a.GetUpperBound(0));
  Console.WriteLine("Upper bound(1) = {0}", a.GetUpperBound(1));
  for (int i = 0; i <= a.GetUpperBound(0); i++)
  {
    for (int j = 0; j <= a.GetUpperBound(1); j++)
    {
      Console.Write(string.Format("{0} ", a[i,j]));
    }
    Console.WriteLine();
  }
}
```

Visual Basic

```
Sub DoRectangularArray()
  Dim a(,) As Integer = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10,11,12}}
  Console.WriteLine("Rank = {0}", a.Rank)
  Console.WriteLine("Length = {0}", a.Length)
  Console.WriteLine("Upper bound(0) = {0}", a.GetUpperBound(0))
  Console.WriteLine("Upper bound(1) = {0}", a.GetUpperBound(1))
  Dim i As Integer
  Dim j As Integer
  For i = 0 To a.GetUpperBound(0)
    For j = 0 To a.GetUpperBound(1)
      Console.Write(String.Format("{0} ", a(i, j)))
    Next
    Console.WriteLine()
  Next
End Sub
```

To declare a rectangular array with explicit dimensions, you just supply the numbers in braces (or parentheses), like this:

C#

```
int[,] a = new int[4,3];
```

Visual Basic

```
Dim a(4,3) as Integer
```

Again, you can provide explicit initialization of such an array in C#, but not in Visual Basic.

The rank of an array is the number of dimensions. A one-dimensional array has a **Rank** value of 1. Each dimension can have a different upper and lower bound. In order to go through the entire array in a nested loop, you must test your loop index variable against the array's upper bound. The **GetUpperBound** method returns the upper bound for a particular rank in the array.

Jagged Arrays

A jagged array is an "array of arrays" that you're familiar with if you've done any C or C++ programming. A two-dimensional jagged array, for example, is a one-dimensional array of one-dimensional arrays. It's not really a multi-dimensional array. The Rank property, for example, is always 1 for a jagged array.

Jagged arrays have certain advantages over rectangular arrays. They can be more space efficient, because not all rows (in a two-dimensional array) must have the same number of columns. In addition, since each row is its own one-dimensional array, you can easily pass an individual row to a method that requires a one-dimensional array parameter.

Finally, early versions of the .NET Framework implement jagged arrays in a much more efficient manner than rectangular arrays.

You'll find working with jagged arrays much easier if you keep in mind that a jagged array is really an array of arrays. The code below shows how to create, initialize, and access a jagged array.

C#

```
static void DoJaggedArray()
{
  int[][] a = new int[][] {
               new int[]{1,2},
               new int[]{3,4,5},
               new int[]{6,7,8,9},
               new int[]{10,11,12}
              };
  Console.WriteLine("Rank = {0}", a.Rank);
  Console.WriteLine("Length = {0}", a.Length);
  Console.WriteLine("Upper bound(0) = {0}", a.GetUpperBound(0));
  for (int i = 0; i < a.Length; i++)
  {
    for (int j = 0; j < a[i].Length; j++)
    {
      Console.Write(string.Format("{0} ", a[i][j]));
    }
    Console.WriteLine();
  }
}
```

Visual Basic

```
Sub DoJaggedArray()
  Dim a1() As Integer = {1, 2}
  Dim a2() As Integer = {3, 4, 5}
  Dim a3() As Integer = {6, 7, 8, 9}
  Dim a4() As Integer = {10, 11, 12}
  Dim a()() As Integer = {a1, a2, a3, a4}
  Console.WriteLine("Rank = {0}", a.Rank)
  Console.WriteLine("Length = {0}", a.Length)
  Console.WriteLine("Upper bound(0) = {0}", a.GetUpperBound(0))
  Dim i, j As Integer
  For i = 0 To a.Length - 1
    For j = 0 To a(i).Length - 1
      Console.Write(String.Format("{0} ", a(i)(j)))
    Next
    Console.WriteLine()
  Next
End Sub
```

I was unable to construct syntax that would initialize a jagged array in Visual Basic as I did in C#. I'm not saying that such syntax doesn't exist. I was just unable to figure it out. Instead, I wrote code that creates four arrays and then creates a fifth array that has those

four arrays as elements. More than anything, this illustrates the jagged array's "array of arrays" nature.

It is possible, by the way, to construct an array that combines both types of dimensions. That is, you can have a two-dimensional array of one-dimensional arrays, like this:

```
int[,][] a;
```

An array of two-dimensional arrays:

```
int[][,] b;
```

Or even a jagged array of two-dimensional rectangular arrays:

```
int[][][,] c;
```

I'll leave the initialization and access of such beasts as an exercise for the reader.

Enumerating Items in an Array

In the preceding examples, I used **for** loops to access each item in the array. There is another way to do it that doesn't require you to know the specific bounds of the array. The **foreach** (**for each** in Visual Basic) statement will determine the array bounds and allow you to process each item in the array. For example, to obtain all of the items in a one-dimensional array, you could write:

C#

```
foreach (int item in a)
{
  Console.WriteLine(item);
}
```

Visual Basic

```
Dim item as Integer
For Each item in a
  Console.WriteLine(item)
Next
```

This is handy for one-dimensional arrays (and can be extended for jagged arrays), and if you want to perform the same operation on all items in a rectangular array. However, **foreach** doesn't really "understand" dimensions—it just processes each element in the array in turn—so it's kind of difficult to do a line break, for example, after each row in a rectangular array.

There is one other caution. **foreach** provides a *reference* to the item in the array or collection that it's enumerating. If the array contains objects, then this isn't a problem,

because when you act on an object reference you're acting on the object itself. But if the array contains value types, the item that **foreach** provides is a *copy* of the original item. If you change the copy, that change isn't propagated in the item stored in the array.

For example, the Visual Basic code below uses a **For** loop to initialize an array, displays the array's contents, and then attempts to set each element to 0 using a **For Each** loop. It is somewhat surprising, then, to find that subsequently displaying the array's contents shows that the items in the array have not changed.

```
Sub ForEachSample()
  Dim a(5)
  Dim i As Integer
  ' initialize the array
  For i = 0 To 5
    a(i) = i
  Next
  ' show its contents
  For Each i In a
    Console.WriteLine(i)
  Next
  ' set each item to 0
  For Each i In a
    i = 0
  Next
  ' and display the contents again
  For Each i In a
    Console.WriteLine(i)
  Next
 End Sub
```

Arrays Miscellanea

Static methods of the Array class provide the ability to sort and reverse arrays, to perform a binary search on a sorted array, and to sequentially search for items in an unsorted array. The **CreateInstance** static method allows you to create arrays of any type with arbitrary lower bounds. If you need such arrays, you should study the sample provided in the MSDN documentation for the CreateInstance method. However, be aware that the Common Language Specification requires that all arrays in CLS-compliant code have a zero lower bound.

The **Array** class implements the **ICloneable, IList, ICollection,** and **IEnumerable** interfaces, which provide some other capabilities. Note that **Array** "implements" some of the IList methods by throwing a **NotImplementedException**. See the documentation for the Array class for more information.

Resizing an Array

Last updated Nov 14, 2003.

To save memory in programs that deal with variable-sized lists, one common technique is to create a small array and then "grow" the array as it fills. Typically, the array is "grown" in chunks rather than one item at a time. This technique gives you a very good tradeoff between performance and memory usage. But an **Array** in .NET is a fixed-capacity construct: once you create it, you can't change its size. What to do?

The trick is to create a new array with the larger capacity, copy the contents of the original array to the new array, and then assign the variable that holds the original array reference to the new array. Remember that, since **Array** is an object, the array variable is really just a reference to the array itself. The following code creates and fills an array, then resizes the array and adds a few more elements.

C#

```
static void DoArrayResize()
{
  // the original array has 10 items
  int[] a = new int[10] {0,1,2,3,4,5,6,7,8,9};

  // we want to resize it to 20 items
  // create the new array
  int[] b = new int[20];

  // copy the contents of the original array to the new array
  a.CopyTo(b, 0);

  // and then assign the new array to the old variable
  a = b;

  // add the new items
  for (int i = 10; i < 20; i++)
  {
    a[i] = i;
  }

  // output array contents
  for (int i = 0; i < a.Length; i++)
    Console.WriteLine(a[i]);
}
```

Visual Basic

```
Sub DoArrayResize()
  ' the original array has 10 items
  Dim a() As Integer = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

  ' we want it to have 20
  ' create the new array
  Dim b(19) As Integer

  ' Copy the contents of the original array to the new array
  a.CopyTo(b, 0)
```

```
  ' and then assign the new array to the old variable
  a = b

  ' add new items to the array
  Dim i As Integer
  For i = 10 To 19
    a(i) = i
  Next

  ' output array contents
  For i = 0 To 19
    Console.WriteLine(a(i))
  Next
End Sub
```

The amount by which you decide to grow the array is quite dependent on your application. Some applications grow by doubling; others grow by adding 50% to the size of the array. Still others use more complicated growth schemes that are optimized for particular situations.

You can use the same technique to shrink an array after deleting items from it.

ArrayList

Last updated Nov 14, 2003.

Think of an **ArrayList** as a resizable one-dimensional array of objects. In addition to automatic resizing, **ArrayList** has a few other advantages over **Array**. **ArrayList** has methods that will add, insert, and remove a range of elements, and it's very easy to create a thread-safe **ArrayList** using the **Synchronized** method.

ArrayList isn't without its disadvantages, the primary one being that **ArrayList** is restricted to a single dimension with a lower bound of zero. Also, because **ArrayList** is limited to storing only **Object**s, a one-dimensional **Array** of a specific type will perform better than an **ArrayList**, especially if the items you're storing are value types that have to be boxed and unboxed.

You'll find **ArrayList** very convenient to use, and it performs well enough for all but the most performance-sensitive applications. Unless you *must* have multiple dimensions, you're probably better off using an **ArrayList** in the initial development of your program. If you then determine that **ArrayList** is causing performance problems, it's a relatively simple matter to modify the code to replace the **ArrayList** with an **Array**.

The **Capacity** property tells you how many items the list can hold before it is resized. You can set the **Capacity** explicitly if you know how many items you need to add to the list. This will increase performance by eliminating the need to resize the list incrementally as items are added.

Count is a read-only property that tells you how many items are currently in the list. **Count** will always be less than or equal to **Capacity**. If **Count** exceeds **Capacity** when you're adding an item to the list, the list is resized. If you try to set **Capacity** to a value that's less than **Count**, **ArrayList** will throw an **ArgumentOutOfRangeException**. The **TrimToSize** method will set **Capacity** equal to **Count**, resizing the list and minimizing memory use.

Using an ArrayList

ArrayList has three constructors. The default, parameterless, constructor creates an empty **ArrayList** that has a default initial capacity of 16 items. There also is a constructor that allows you to specify the list's initial capacity. The final constructor takes an **ICollection** interface reference, and copies all the items from the collection into the new **ArrayList**. The **Count** and **Capacity** properties of the new list are set to reflect the number of items copied. Here are some examples:

C#

```
static void DoArrayList()
{
  // create an ArrayList and set its Capacity to 100
  ArrayList al1 = new ArrayList();
  al1.Capacity = 100;

  // create an ArrayList with the initial Capacity of 100
  ArrayList al2 = new ArrayList(100);

  // create an ArrayList from the passed Array
  int[] a = new int[] {0,1,2,3,4,5,6,7,8,9};
  ArrayList al3 = new ArrayList(a);

  foreach (object o in al3)
  {
    Console.WriteLine(o.ToString());
  }
}
```

Visual Basic

```
Sub DoArrayList()
  ' create an ArrayList and set its Capacity to 100
  Dim al1 As ArrayList = New ArrayList
  al1.Capacity = 100

  ' create an ArrayList with an initial Capacity of 100
  Dim al2 As ArrayList = New ArrayList(100)

  ' create an Arraylist from the passed Array
  Dim a() As Integer = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  Dim al3 As ArrayList = New ArrayList(a)

  Dim o As Object
```

```
  For Each o In al3
    Console.WriteLine(o.ToString())
  Next
End Sub
```

The **Add** method adds an item to the end of the **ArrayList**. **Insert** will insert an item into the list at the specified index, moving all items beyond it "down" one place in the list. **AddRange** and **InsertRange** will add or insert multiple items from an **ICollection** into the list.

Two methods can remove a single item from an **ArrayList**. **Remove** removes the first object in the list (searched in sequential order) that is equal to the object passed. Equality is determined by calling **Object.Equals**. **RemoveAt** removes the item at the specified index in the list. In both cases, the item is removed from the list and all subsequent items are moved up one place in the list.

The **RemoveRange** method removes a range of items from the list, starting at the supplied index and removing the specified count of items. All subsequent items in the list are moved up *count* places in the list.

I suspect an example will help clarify things. The code below shows how to manipulate the **ArrayList** using the **Add**, **Insert**, and **Remove** methods and some variants.

C#

```
static void OutputList(ArrayList al)
{
  Console.WriteLine("-----");
  foreach (Object o in al)
    Console.WriteLine(o.ToString());
}

static void DoArrayList()
{
  // create an ArrayList
  ArrayList al = new ArrayList();

  // add a couple of items
  al.Add (1);
  al.Add (2);

  // insert an item between the two
  al.Insert(1, 191);

  OutputList(al);

  // insert a range of items into the list
  int[] a = new int[] {-1, -2, -3, -4, -5, -6};
  al.InsertRange(2, a);

  OutputList(al);
```

```
  // remove an item from the list
  al.Remove(-4);

  // remove an item from a specific place in the list
  al.RemoveAt(5);

  // remove items 2, 3, and 4 (start at index 2 and delete 3 items)
  al.RemoveRange(2, 3);

  OutputList(al);
}
```

Visual Basic

```
Sub DoArrayList()
  ' create an ArrayList
  Dim al As ArrayList = New ArrayList

  ' Add some items
  al.Add(1)
  al.Add(2)

  ' insert an item between the two
  al.Insert(1, 191)

  OutputList(al)

  ' insert a range of items into the list
  Dim a() As Integer = {-1, -2, -3, -4, -5, -6}
  al.InsertRange(2, a)

  OutputList(al)

  ' remove an item from the list
  al.Remove(-4)

  ' remove an item from a specific place in the list
  al.RemoveAt(5)

  ' remove items 2, 3, and 4 (start at index 2 and delete 3 items)
  al.RemoveRange(2, 3)

  OutputList(al)
End Sub
```

You access individual items in the ArrayList through the **Item** property. For example:

C#

```
Object o = al.Item[3];
```

Visual Basic

```
Dim o as Object = al.Item(3)
```

Item is the indexer for the **ArrayList** class, so in C# and Visual Basic (and any other .NET language that supports indexers), you can access items in the **ArrayList** using array-like syntax. For example, the code below sets the fourth item in the list:

C#

```
al[3] = new Object();
```

Visual Basic

```
al(3) = new Object()
```

Like **Array**, **ArrayList** implements a number of interfaces that are common to all collection types, allowing you to in many circumstances treat an **ArrayList** like a generic list or collection. The interfaces that **ArrayList** implements are **IList**, **ICollection**, **ICloneable**, and **IEnumerable**. Check the online reference for each of those interfaces for more information on their capabilities.

Array and **ArrayList** are the workhorses of .NET collections. Most of the advanced collection types (**HashTable**, **SortedList**, **Queue**, etc.) use one or the other of these base collection types as their underlying data structure. Understanding how **Array** and **ArrayList** work will get you a long way towards understanding all of the .NET collection types.

Searching for Items in Collections

Last updated Nov 21, 2003.

Both **Array** and **ArrayList** provide methods that allow you to search the collection sequentially for items. Both classes implement the **IndexOf** and **LastIndexOf** methods. **IndexOf** returns the zero-based index of the first item in the list (searched sequentially) that matches the passed argument. If the item is not found in the collection, **IndexOf** returns –1. For example, the following code shows how to use **IndexOf** to search in an **Array** and in an **ArrayList**.

C#

```
static void DoIndexOf()
{
  // build and search an array
  string[] a = new string[] {"the", "hand", "is",
    "quicker", "than", "the", "eye"};
  int i = Array.IndexOf(a, "hand");
  Console.WriteLine("index of 'hand' = {0}", i);
  i = Array.IndexOf(a, "ear");
  Console.WriteLine("index of 'ear' = {0}", i);

  // now search an ArrayList
  ArrayList al = new ArrayList(a);
```

```
  Console.WriteLine("index of 'hand' = {0}",
    al.IndexOf("hand"));
  Console.WriteLine("index of 'ear' = {0}",
    al.IndexOf("ear"));
}
```

Visual Basic

```
Sub DoIndexOf()
  ' build and search an array
  Dim a() As String = New String() {"the", "hand", "is", _
    "quicker", "than", "the", "eye"}
  Dim i As Integer = Array.IndexOf(a, "hand")
  Console.WriteLine("index of 'hand' = {0}", i)
  i = Array.IndexOf(a, "ear")
  Console.WriteLine("index of 'ear' = {0}", i)

  ' now search an ArrayList
  Dim al As ArrayList = New ArrayList(a)
  Console.WriteLine("index of 'hand' = {0}", _
    al.IndexOf("hand"))
  Console.WriteLine("index of 'ear' = {0}", _
    al.IndexOf("ear"))
End Sub
```

Notice that **IndexOf** is a static (Shared in Visual Basic) method of the **Array** class, but an instance method of the **ArrayList** class. Other than that detail, the methods work exactly the same for both classes.

There are two overloaded versions of **IndexOf** that will search sections of the list rather than the entire list. One overloaded method searches from the specified index to the end of the list, and the other searches a subset of the list that is identified by a starting index and a number of items. These two overloads can be quite useful. For example, suppose you wanted to find all occurrences of "the" in the list of words shown above:

C#

```
// search for all occurrences of "the"
i = Array.IndexOf(a, "the");
while (i != -1)
{
  Console.WriteLine("index of 'the' = {0}", i);
  // search for next occurrence
  i = Array.IndexOf(a, "the", i + 1);
}
```

Visual Basic

```
' search for all occurrences of "the"
i = Array.IndexOf(a, "the")
While i <> -1
  Console.WriteLine("index of 'the' = {0}", i)
```

```
  ' search for next occurrence
  i = Array.IndexOf(a, "the", i + 1)
End While
```

The **LastIndexOf** method works in much the same way as **IndexOf**, but searches the list backwards.

Comparing Items

Last updated Nov 21, 2003.

IndexOf and **LastIndexOf** use the **Object.Equals** method to determine whether two items are equal. For non-intrinsic (that is, user-defined) types, the **Equals** implementation for that type is used. If you'll be searching a collection for items of your user-defined types, be sure to override the **Equals** method in the type. Otherwise, the collection will use the default **Object.Equals** method, which simply compares the addresses of the items. For example, consider this simple class:

```
public class Person
{
  private string firstName;
  private string lastName;
  public Person(string fname, string lname)
  {
    firstName = fname;
    lastName = lname;
  }
  // code that implements properties...
}
```

If you create two "identical" objects and compare them as in the code below, you might be surprised to find that they don't compare as equal.

```
static void DoCompare()
{
  Person jim1 = new Person("Jim", "Mischel");
  Person jim2 = new Person("Jim", "Mischel");

  if (jim1 == jim2)
    Console.WriteLine("equal");
  else
    Console.WriteLine("not equal");
}
```

Since you didn't tell .NET how to compare the items, it used the default **Object.Equals** method, which simply compares the addresses. Obviously, two different objects can't share the same address, so the result is "not equal."

To make this work correctly, you need to override **Equals** for user-defined types. In C#, you'll also need to overload the equality operator (==) and the corresponding inequality operator (!=). The code below shows how that's done.

C#

```
public class Person
{
  private string firstName;
  private string lastName;
  public Person(string fname, string lname)
  {
    firstName = fname;
    lastName = lname;
  }

  public static bool operator ==(Person p1, Person p2)
  {
    return (p1.firstName == p2.firstName) &&
      (p1.lastName == p2.lastName);
  }

  public static bool operator !=(Person p1, Person p2)
  {
    return !(p1 == p2);
  }

  // provide an Equals method
  public override bool Equals(Object obj)
  {
    // Check for null values and compare run-time types.
    if (obj == null || GetType() != obj.GetType())
      return false;
    return this == (Person)obj;
  }

  // provide a GetHashCode method
  public override int GetHashCode()
  {
    string s = firstName + lastName;
    return s.GetHashCode();
  }
  // code that implements properties...
}
```

Visual Basic

```
Public Class Person
  Dim firstName As String
  Dim lastName As String

  Public Sub New(ByVal fname As String, ByVal lname As String)
    firstName = fname
    lastName = lname
  End Sub
```

```
  Public Overloads Overrides Function Equals(ByVal obj As Object) As
Boolean
    If obj Is Nothing Or Not Me.GetType() Is obj.GetType() Then
      Return False
    End If
    Dim p As Person = CType(obj, Person)
    Return Me.firstName = p.firstName And Me.lastName = p.lastName
  End Function

  Public Overrides Function GetHashCode() As Integer
    Dim s As String = firstName + lastName
    Return s.GetHashCode()
  End Function
End Class

Sub DoCompare()
  Dim jim1 As Person = New Person("jim", "mischel")
  Dim jim2 As Person = New Person("jim", "mischel")

  If jim1.Equals(jim2) Then
    Console.WriteLine("equal")
  Else
    Console.WriteLine("not equal")
  End If
End Sub
```

In accordance with the Guidelines for Implementing Equals and the Equality Operator (==), I also implemented the **GetHashCode** method. **GetHashCode** is used by the **HashTable** and other advanced collection types. Also in accordance with the guidelines, I created an overloaded equality operator and implemented **Equals** such that **Equals** uses == to do the comparison. It's not possible to override the equality operator in Visual Basic.

ArrayList has an additional method, **Contains**, which simply tells you whether a particular item exists in the list. It, too, depends on the object's **Equals** method to determine equality.

Sorting and Binary Search

Last updated Jan 1, 2004.

Although the .NET Framework provides some ordered collection classes (**HashTable** and **SortedList**, for example), it's often useful to sort an **Array** or **ArrayList**, and use a binary search to locate items. Considering that a sequential search takes, on the average, N/2 comparisons to find an record in a list of N items, and a binary search takes only about $\log_2(N)$ comparisons, it makes a lot of sense to use a binary search whenever possible. In a list of a million items, a sequential search will take an average of 500,000 comparisons to find a particular item. A binary search will take a maximum of 20. If you have a large list that doesn't change very often, or that changes infrequently in

comparison to searches, then it's a good idea to keep it sorted so that you can enjoy the performance advantages of the binary search.

Both **Array** and **ArrayList** implement **Sort** and **BinarySearch** methods that you can use to sort and search lists. As with the **IndexOf** method, the **Array** versions of these methods are static (Shared in Visual Basic).

Simple Sorting

Last updated Jan 1, 2004.

There are several ways to sort the items in **Array** and **ArrayList** objects. The most basic sort compares objects directly using the **IComparable** interface that's implemented by the individual objects. For example, the code below sorts an **Array** of strings, and an **ArrayList** that contains the same strings.

C#

```
static void DoSorts()
{
  string[] ituPhonetics =
    new string[] {
              "Bravo",
              "Foxtrot",
              "Echo",
              "Alpha",
              "Delta",
              "Charlie"
            };
  ArrayList al = new ArrayList(ituPhonetics);

  // sort the array
  Console.WriteLine("----------");
  Array.Sort(ituPhonetics);
  foreach (string s in ituPhonetics)
    Console.WriteLine(s);

  // sort the ArrayList
  Console.WriteLine("----------");
  al.Sort();
  foreach (string s in al)
    Console.WriteLine(s);
}
```

Visual Basic

```
Sub DoSorts()
  Dim ituPhonetics() As String = New String() { _
    "Bravo", "Foxtrot", "Echo", _
    "Alpha", "Delta", "Charlie"}
  Dim al As ArrayList = New ArrayList(ituPhonetics)
```

```
    ' sort the array
    Console.WriteLine("----------")
    Array.Sort(ituPhonetics)
    Dim s As String
    For Each s In ituPhonetics
      Console.WriteLine(s)
    Next

    ' sort the ArrayList
    Console.WriteLine("----------")
    al.Sort()
    For Each s In al
      Console.WriteLine(s)
    Next
End Sub
```

This version of **Sort** calls the individual list items' **CompareTo** method (the only member of the **IComparable** interface). **CompareTo** compares two objects and returns an integer that is:

- Less than zero if the instance item is less than the item it's being compared to;
- Zero if the instance item is equal to the item it's being compared to;
- Greater than zero if the instance item is greater than the item it's being compared to.

In Visual Basic, you can compare two **String** objects using the comparison operators. The semantics of the Visual Basic language "overloads" those operators for **String** objects. For example, the code below will write the message "hello" to the screen:

```
If "abc" < "def" Then Console.WriteLine("hello")
```

That has the same effect as this code that calls the **CompareTo** method:

```
If "abc".CompareTo("def") Then Console.WriteLine("hello")
```

C# doesn't automatically "overload" the comparison objects for **String**s, and those operators aren't overloaded by the implementation of the **String** object. As a result, you can't use them to compare strings. That is, the following won't compile in C#:

```
if ("abc" < "def") Console.WriteLine("hello");
```

In C#, you must use the **CompareTo** method to compare objects unless there are explicit overloads of the comparison operators. The C# code to compare two strings, then, is:

```
if ("abc".CompareTo("def") < 0) Console.WriteLine("hello");
```

The **IComparable** interface is implemented by all of the .NET objects and value types that it makes sense to compare. That includes intrinsic types like **Integer** and **Double**, enumerations, **DateTime**, and many different objects. But if you create your own classes to maintain in a sorted collection, that class must implement the **IComparable** interface

(that is, the **CompareTo** method). The MSDN Library entry for the IComparable.CompareTo method has a good example of implementing this interface for an object.

Using a Different Comparison Method

Last updated Jan 1, 2004.

There will come a time when the objects that you're putting in the list either don't implement **IComparable**, or the objects you're putting in the list are of different types and their **CompareTo** methods don't know how to work together. In that case, you have to create an object that has a method which knows how to compare the different kinds of objects that you're putting into the list.

Or, suppose you want to do a custom sort of a list of **String** objects. For example, you know that all the strings in the list are part numbers with a three-letter prefix and a 4-digit part number; you want to sort them by the 4-digit part number.

The **String** object's **CompareTo** method is no help here. You have to create an object that implements the **IComparer** interface, and pass a reference to that object to the **Sort** method. The code below shows how that's done with an **Array**. The **ArrayList** code is identical, except of course that **Sort** is an instance method rather than a static method.

C#

```
// defines the class that knows how to compare
// two part number strings
public class PartNumberComparer: IComparer
{
  int IComparer.Compare(object x, object y)
  {
    string s1 = (string)x;
    string s2 = (string)y;
    return s1.Substring(3, 4).CompareTo(s2.Substring(3, 4));
  }
}

static void DoComparer()
{
  string[] Parts = new string[]
    {"qed0234", "abc9927", "zzz0015", "xyz6874"};

  // create an instance of the comparer class
  PartNumberComparer myComparer = new PartNumberComparer();

  // sort the array of part numbers using the comparer
  Array.Sort(Parts, myComparer);
  foreach (string s in Parts)
    Console.WriteLine(s);
}
```

Visual Basic

```
' defines the class that knows how to compare
' two part number strings
Public Class PartNumberComparer
  Implements IComparer

  Public Function Compare(ByVal x As Object, ByVal y As Object)
➥ As Integer Implements System.Collections.IComparer.Compare
    Dim s1 As String = x
    Dim s2 As String = y
    Return s1.Substring(3, 4).CompareTo(s2.Substring(3, 4))
  End Function
End Class

Sub DoComparer()
  Dim Parts() As String = _
    {"qed0234", "abc9927", "zzz0015", "xyz6874"}

  ' create an instance of the comparer class
  Dim myComparer As PartNumberComparer = New PartNumberComparer

  ' sort the array of part numbers using the comparer
  Array.Sort(Parts, myComparer)
  Dim s As String
  For Each s In Parts
    Console.WriteLine(s)
  Next
End Sub
```

If you run that code, you'll see that the part numbers are output in numerical order, ignoring the 3-letter prefix.

Array and **ArrayList** share one other version of **Sort**: a method that will sort a specified section of the list. The C# method declarations for the functions are shown below, the **static** function being for **Array**, and the other for **ArrayList**.

```
public static void Sort(
  Array array,
  int index,
  int length,
  IComparer comparer
);
public virtual void Sort(
  int index,
  int count,
  IComparer comparer
);
```

So, to sort the first 10 items in an array (assuming you knew that there were at least 10 items), the C# code would be:

```
Array.Sort(myArray, 0, 10, myComparer);
```

The Visual Basic code would be the same -- minus the semicolon, of course.

Key-sorting an Array

Last updated Nov 26, 2003.

Array has more sorting options than **ArrayList**. **Array** can sort a subset of the list without having to specify a comparer, and **Array** can perform a key sort using two arrays.

It works this way. Assume you have two arrays. One array contains repair parts objects that are available—sort of a catalog from the original manufacturer. The second array is just a list of part numbers. Not the manufacturer's part numbers, but the part numbers that you list in *your* catalog. You're a reseller of the original manufacturer's parts, but your inventory system requires that part numbers have your own special format. (Don't think this is just a contrived example. This kind of thing goes on all the time!)

Now, you want to sort the parts by *your* part number and keep the two arrays in sync. Enter the keyed sort method. The code below shows how it's done.

C#

```
// the class that defines the manufacturer's part information
public class MfgPart: IComparable
{
  private string partNo;
  // other stuff in part: description, price, etc.

  public MfgPart(string pn)
  {
    partNo = pn;
  }
  public string PartNo
  {
    get { return partNo; }
  }
  public int CompareTo (object obj)
  {
    MfgPart p = (MfgPart)obj;
    return this.partNo.CompareTo(p.partNo);
  }
}

// outputs the parts cross reference
static void OutputParts(MfgPart[] rp, string[] mp)
{
  Console.WriteLine("--------");
  for (int i = 0; i < rp.Length; i++)
    Console.WriteLine("{0} - {1}", rp[i].PartNo, mp[i]);
}

static void DoKeySort()
```

```
{
  // manufacturer part information
  MfgPart[] repairParts = new MfgPart[]
  {
    new MfgPart("xqj57"),
    new MfgPart("widget36"),
    new MfgPart("DooDad1"),
    new MfgPart("xyzzy")
  };

  // in-house part numbers match manufacturer parts
  string[] myParts = new string[]
    {"qed0234", "abc9927", "zzz0015", "xyz6874"};

  // output the unsorted cross-reference
  OutputParts(repairParts, myParts);

  // now sort in order by mfg part number
  Array.Sort(repairParts, myParts);

  // output cross-reference sorted by mfg part number
  OutputParts(repairParts, myParts);

  // sort in order by our part number
  PartNumberComparer myComparer = new PartNumberComparer();
  Array.Sort(myParts, repairParts, myComparer);

  // and output cross-reference sorted by internal part number
  OutputParts(repairParts, myParts);
}
```

Visual Basic

```
' the class that defines the manufacturer's part information
Public Class MfgPart
  Implements IComparable

  Dim pNo As String

  Public Sub New(ByVal pn As String)
    pNo = pn
  End Sub

  ReadOnly Property PartNo()
    Get
      Return pNo
    End Get
  End Property

  Public Function CompareTo(ByVal obj As Object) As Integer
➥ Implements System.IComparable.CompareTo
    Dim p As MfgPart = obj
    Return Me.pNo.CompareTo(p.pNo)
  End Function
End Class
```

```
' outputs the parts cross reference
Sub OutputParts(ByVal rp() As MfgPart, ByVal mp() As String)
  Console.WriteLine("----------")
  Dim i As Integer
  For i = 0 To rp.Length - 1
    Console.WriteLine("{0} - {1}", rp(i).PartNo, mp(i))
  Next
End Sub

Sub DoKeySort()
  ' manufacturer part information
  Dim repairParts() As MfgPart = _
  { _
    New MfgPart("xqj57"), _
    New MfgPart("widget36"), _
    New MfgPart("DooDad1"), _
    New MfgPart("xyzzy") _
  }

  ' in-house part numbers match manufacturer parts
  Dim myParts() As String = _
    {"qed0234", "abc9927", "zzz0015", "xyz6874"}

  ' output unsorted cross-reference
  OutputParts(repairParts, myParts)

  ' sort by mfg part number
  Array.Sort(repairParts, myParts)

  ' output cross-reference sorted by mfg part number
  OutputParts(repairParts, myParts)

  ' sort by our part number
  Dim myComparer As PartNumberComparer = New PartNumberComparer
  Array.Sort(myParts, repairParts, myComparer)

  ' output cross-reference sorted by internal part number
  OutputParts(repairParts, myParts)
End Sub
```

Here's the output from the program:

```
----------
xqj57 - qed0234
widget36 - abc9927
DooDad1 - zzz0015
xyzzy - xyz6874
----------
DooDad1 - zzz0015
widget36 - abc9927
xqj57 - qed0234
xyzzy - xyz6874
----------
DooDad1 - zzz0015
xqj57 - qed0234
xyzzy - xyz6874
```

```
widget36 - abc9927
```

The first four lines just output the list as it was originally built, showing the correspondence between the manufacturer's part numbers on the left and our part numbers on the right. The second group of lines shows the list sorted alphabetically by manufacturer's part number. Note that the corresponding internal part numbers (our numbers) remain. A *DooDad1* is still internal part number *zzz0015*. The last group shows the same parts, but this time sorted by internal part number using the **PartNumberComparer** class.

Key-sorting an Array

Last updated Nov 26, 2003.

Array has more sorting options than **ArrayList**. **Array** can sort a subset of the list without having to specify a comparer, and **Array** can perform a key sort using two arrays.

It works this way. Assume you have two arrays. One array contains repair parts objects that are available—sort of a catalog from the original manufacturer. The second array is just a list of part numbers. Not the manufacturer's part numbers, but the part numbers that you list in *your* catalog. You're a reseller of the original manufacturer's parts, but your inventory system requires that part numbers have your own special format. (Don't think this is just a contrived example. This kind of thing goes on all the time!)

Now, you want to sort the parts by *your* part number and keep the two arrays in sync. Enter the keyed sort method. The code below shows how it's done.

C#

```
// the class that defines the manufacturer's part information
public class MfgPart: IComparable
{
  private string partNo;
  // other stuff in part: description, price, etc.

  public MfgPart(string pn)
  {
    partNo = pn;
  }
  public string PartNo
  {
    get { return partNo; }
  }
  public int CompareTo (object obj)
  {
    MfgPart p = (MfgPart)obj;
    return this.partNo.CompareTo(p.partNo);
  }
}
```

```
// outputs the parts cross reference
static void OutputParts(MfgPart[] rp, string[] mp)
{
  Console.WriteLine("--------");
  for (int i = 0; i < rp.Length; i++)
    Console.WriteLine("{0} - {1}", rp[i].PartNo, mp[i]);
}

static void DoKeySort()
{
  // manufacturer part information
  MfgPart[] repairParts = new MfgPart[]
  {
    new MfgPart("xqj57"),
    new MfgPart("widget36"),
    new MfgPart("DooDad1"),
    new MfgPart("xyzzy")
  };

  // in-house part numbers match manufacturer parts
  string[] myParts = new string[]
    {"qed0234", "abc9927", "zzz0015", "xyz6874"};

  // output the unsorted cross-reference
  OutputParts(repairParts, myParts);

  // now sort in order by mfg part number
  Array.Sort(repairParts, myParts);

  // output cross-reference sorted by mfg part number
  OutputParts(repairParts, myParts);

  // sort in order by our part number
  PartNumberComparer myComparer = new PartNumberComparer();
  Array.Sort(myParts, repairParts, myComparer);

  // and output cross-reference sorted by internal part number
  OutputParts(repairParts, myParts);
}
```

Visual Basic

```
' the class that defines the manufacturer's part information
Public Class MfgPart
  Implements IComparable

  Dim pNo As String

  Public Sub New(ByVal pn As String)
    pNo = pn
  End Sub

  ReadOnly Property PartNo()
    Get
      Return pNo
```

```
    End Get
  End Property

  Public Function CompareTo(ByVal obj As Object) As Integer
➥ Implements System.IComparable.CompareTo
    Dim p As MfgPart = obj
    Return Me.pNo.CompareTo(p.pNo)
  End Function
End Class

' outputs the parts cross reference
Sub OutputParts(ByVal rp() As MfgPart, ByVal mp() As String)
  Console.WriteLine("----------")
  Dim i As Integer
  For i = 0 To rp.Length - 1
    Console.WriteLine("{0} - {1}", rp(i).PartNo, mp(i))
  Next
End Sub

Sub DoKeySort()
  ' manufacturer part information
  Dim repairParts() As MfgPart = _
  { _
    New MfgPart("xqj57"), _
    New MfgPart("widget36"), _
    New MfgPart("DooDad1"), _
    New MfgPart("xyzzy") _
  }

  ' in-house part numbers match manufacturer parts
  Dim myParts() As String = _
    {"qed0234", "abc9927", "zzz0015", "xyz6874"}

  ' output unsorted cross-reference
  OutputParts(repairParts, myParts)

  ' sort by mfg part number
  Array.Sort(repairParts, myParts)

  ' output cross-reference sorted by mfg part number
  OutputParts(repairParts, myParts)

  ' sort by our part number
  Dim myComparer As PartNumberComparer = New PartNumberComparer
  Array.Sort(myParts, repairParts, myComparer)

  ' output cross-reference sorted by internal part number
  OutputParts(repairParts, myParts)
End Sub
```

Here's the output from the program:

```
----------
xqj57 - qed0234
widget36 - abc9927
DooDad1 - zzz0015
```

```
xyzzy - xyz6874
----------
DooDad1 - zzz0015
widget36 - abc9927
xqj57 - qed0234
xyzzy - xyz6874
----------
DooDad1 - zzz0015
xqj57 - qed0234
xyzzy - xyz6874
widget36 - abc9927
```

The first four lines just output the list as it was originally built, showing the correspondence between the manufacturer's part numbers on the left and our part numbers on the right. The second group of lines shows the list sorted alphabetically by manufacturer's part number. Note that the corresponding internal part numbers (our numbers) remain. A *DooDad1* is still internal part number *zzz0015*. The last group shows the same parts, but this time sorted by internal part number using the **PartNumberComparer** class.

Dictionary Collection Types

Last updated Dec 5, 2003.

Simple lists like **Array** and **ArrayList** are fine for many purposes, but become difficult to use, or slow, when the program is continually modifying the list and searching for items. Simple collections must be maintained in sorted order, or searched sequentially. Both options are computationally expensive.

A more efficient and easier-to-use option is a dictionary-based approach, in which items in the collection consist of a key and value pair. The collection machinery maintains an efficient data structure that allows rapid insertion, deletion, and searching of keys. One such data structure is called a *hash table*.

A hash table uses a *hash function* to generate a numeric value based on the value of the key that you provide to it. The result returned by the hash function is called the *hash* or *hash code* of the passed record. This value is then stored and used to quickly locate a particular item in the collection.

A hash function might not return a unique number for every record. That is, it's possible that the result of the hash function for two different strings would be the same. When this occurs, it is called a *hash clash*. The code that uses the hashing function must detect and properly handle this condition. A hash table will be much more efficient, though, if you carefully select a hash function that will minimize clashes.

The .NET Framework provides two dictionary-based collection types, both based on the **IDictionary** interface. Those collections are **Hashtable** and **SortedList**.

Hashtable

Last updated Dec 5, 2003.

The **Hashtable** collection type stores key-value pairs and allows you to locate items in the collection based on the key. For example, suppose you want to store the names and titles of people in the U.S. Government's Executive Branch. You could represent such information in a table like this (abbreviated for obvious reasons):

Title	Name
President	George W. Bush
Vice President	Dick Cheney
Secretary of State	Colin Powell

If you want to know who the Secretary of State is, you just read down the left column to find that title, and read across to get the name. A **Hashtable** works in much the same way. When you add an item, you supply the key and the value. You can then reference items by value. The code below shows how it's done.

C#

```
static void DoHashtable()
{
  // create the hash table
  Hashtable execBranch = new Hashtable();

  // add items
  execBranch.Add("President", "George W. Bush");
  execBranch.Add("Vice President", "Dick Cheny");
  execBranch.Add("Secretary of State", "Colin Powell");

  // retrieve vice president's name based on title
  string VP = (string)execBranch["Vice President"];
  Console.WriteLine("Vice President is: {0}", VP);
}
```

Visual Basic

```
Sub DoHashtable()
  ' create the hash table
  Dim execBranch As Hashtable = New Hashtable

  ' add items
  execBranch.Add("President", "George W. Bush")
  execBranch.Add("Vice President", "Dick Cheny")
  execBranch.Add("Secretary of State", "Colin Powell")

  ' retrieve vice president's name based on title
```

```
  Dim VP As String = execBranch("Vice President")
  Console.WriteLine("Vice President is {0}", VP)
End Sub
```

The **Add** method, by the way, will throw an exception if an item with the specified key value already exists in the collection. You can't use **Hashtable** to store multiple items with the same key.

When you're looking up items by key, if there is no matching key in the **Hashtable**, the accessor method will return **null** (**Nothing** in Visual Basic). For example, if the code that was looking for the Vice President's name was written like this:

```
string VP = execBranch["VicePres"];
```

then the **VP** variable would be set to null.

You can remove an item from the **Hashtable** by passing the key of the item to remove to the **Remove** method.

You can enumerate all of the items in the **Hashtable** by obtaining the list's enumerator object and iterating through it, like this:

C#

```
// output all keys and values
IDictionaryEnumerator myEnumerator = execBranch.GetEnumerator();
while (myEnumerator.MoveNext())
{
  Console.WriteLine("{0}, {1}", myEnumerator.Key, myEnumerator.Value);
}
```

Visual Basic

```
' output all keys and values
Dim myEnumerator As IDictionaryEnumerator = execBranch.GetEnumerator()
While myEnumerator.MoveNext()
  Console.WriteLine("{0}, {1}", myEnumerator.Key, myEnumerator.Value)
End While
```

You can also obtain all of the keys by examining the **Hashtable**'s **Keys** property, and all of the values by examining the **Values** property. Both of these properties are **ICollection** types.

Hashtable Construction Options

Last updated Dec 5, 2003.

The easiest way to create a **Hashtable** is by calling the default constructor, as shown in the code sample above. This call creates a **Hashtable** collection with default options that

make the best balance between speed, size, and functionality. However, in special circumstances you might want to change the way that the collection is created. The various overloaded **Hashtable** constructors let you specify several parameters, as outlined below.

- Specifying an *initial capacity* initializes the **Hashtable** to hold a certain number of items. This saves some time in resizing the collection as items are added. This is very beneficial if you have a general idea how many items will go into the collection.
- The *load factor* specifies a ratio between elements and buckets in the Hashtable. Without going into the detail of buckets (see the official documentation for details), a lower number here gives faster performance at the expense of increased memory consumption. The number you supply here can be between 0.1 and 1.0. A value of 1.0 gives the best balance between speed and memory consumption.
- The default constructor has the **Hashtable** use the individual keys' **GetHash** function to obtain hash codes for keys. To override that behavior, you can pass a reference to an object that implements the **IHashCodeProvider** interface. The **Hashtable** will then use this interface to obtain hash codes for keys. In collections derived from **Hashtable**, you can also change the hash code provider by setting the protected **hcp** property.
- By default, Hashtable compares keys by calling individual key objects' **Equals** method. You can override this by passing an **IComparer** reference to the constructor. In addition, derived objects can change the comparison function by setting the protected **comparer** property.

The **Hashtable** is a very powerful collection type that is used in many places throughout the .NET Framework. If you find that you need to store and access objects based on a key, the **Hashtable** is the first place you should turn. This collection type has other methods and properties that I did not discuss here. You should refer to the official documentation for more details.

SortedList

Last updated Jan 1, 2004.

The **SortedList** collection type is something of a hybrid of **Hashtable** and **Arraylist**. Like **Hashtable**, **SortedList** is based on the **IDictionary** interface, so every element is a key-and-value pair and can be accessed by the key. **SortedList** is like **ArrayList** in that it is a sequence of elements that can be accessed by the value or by the index. The list is maintained in sorted order based on a specified comparer.

Use **SortedList** when you want a collection that stores key-and-value pairs, and you also want the flexibility of an indexed list.

You access items in the **SortedList** using the same techniques that were described for **Hashtable** and **ArrayList**. Like **Hashtable**, the default sorting method is to use the

IComparable interface defined by each key object. You can change that behavior by calling the overloaded **SortedList** constructor that takes a reference to an **IComparer** interface. Unlike **Hashtable**, however, there is no way to change the comparison method after you create the list, and you cannot change the way that hash codes are generated. In fact, the documentation doesn't specify that **SortedList** even *uses* hash codes for the keys.

You can add items to the **SortedList** by calling the **Add** method as with **Hashtable**, or by setting the value of a non-existent key. For example, this C# code will create a new entry in the **SortedList**.

```
mySortedList["NewKey"] = myObject;
```

If no element with the key value "NewKey" exists in the list, then a new element is added to the list. If such an element already exists, then **myObject** will overwrite the previous value for that entry.

The **Add** method, on the other hand, throws an exception if you attempt to add an element with a key that already exists in the list. Keys must be unique within the list, but values don't have to be. It's perfectly okay to write:

```
mySortedList.Add("NewKey1", myObject);
mySortedList.Add("NewKey2", myObject);
```

Although you probably should have a very good reason for adding a single object twice to the same list.

Queue and Stack

Last updated Jan 1, 2004.

The **Queue** and **Stack** collection types implement the common data structures of the same name. A queue is a first-in, first-out data structure, like a line at a grocery store. Items go in at the tail of the queue, and are removed from the head. The **Enqueue** method adds an item to the queue, and **Dequeue** removes the oldest element (that is, the one that's been in the queue the longest). **Peek** is a little cheat that lets you examine the item at the front of the queue without removing it.

A stack is a first in, last out data structure, like the pile on my desk or the call stack in a microprocessor. Items are added to the top of the stack, pushing all the other items down. Items are removed from the top of the stack in reverse order. The **Push** method places an item on the stack and **Pop** removes an item. Like **Queue**, **Stack** implements **Peek** so that you can look at the top-of-stack item without having to remove it from the collection.

As with the other collection types, **Queue** and **Stack** implement the **Count** property, so you can tell how many items they contain. They also have various methods to copy the

contents of the collection to an **Array**. Overloaded methods allow you to initialize the collection with a specific capacity or with the items from another collection.

Using Collections

When deciding which collection type to use in your application, you need to consider many factors. Choosing the wrong collection type will hamper your ability to access data, and can cause severe performance problems. Using an **Array**, for example, when you need to access items by key or maintain items in sorted order, will cause you to write a lot of extraneous code that searches or sorts the items. The Grouping Data in Collections topic in MSDN has some good guidelines for selecting a collection class. In this section, I will provide examples of using some of the Framework-supplied collection classes.

Collections and Thread Safety

Last updated Jan 1, 2004.

The default behavior of all the collection classes is not generally thread safe. Although any number of threads can *read* a collection simultaneously, a single thread modifying the collection can cause undefined behavior for any other threads that are accessing it. The **Hashtable** class guarantees thread safety for a single writer and multiple readers, but if multiple writers are required, the same rules apply as for all the other collection types.

The .NET Framework SDK documentation identifies three primary ways to make collections thread safe:

- Create a thread-safe wrapper using the **Synchronized** method, and access the collection exclusively through that wrapper.
- If the class does not have a **Synchronized** method, derive from the class and implement a **Synchronized** method using the **SyncRoot** property.
- Use a locking mechanism, such as the **lock** statement in C# (**SyncLock** in Visual Basic), on the **SyncRoot** property when accessing the collection.

lock (**SyncLock** in Visual Basic) is the easiest and most effective way to guarantee thread safety. Lock is most useful when enumerating a collection. Enumerating through a collection is inherently not thread-safe, and will throw an exception if the collection is modified at any point during the enumeration. To prevent any other thread from modifying the collection, you lock the collection first, like this:

C#

```
SortedList myCollection = new SortedList();
// code that adds items to the collection
// lock the collection and enumerate it
lock (myCollection.SyncRoot)
```

```
{
  foreach (Object item in myCollection)
  {
    // insert your enumeration code here
  }
}
```

Visual Basic

```
Dim myCollection as New SortedList()
' code that adds items to the collection
' lock the collection and enumerate it
Dim item as Object
SyncLock myCollection.SyncRoot
  For Each item In myCollection
    ' insert your enumeration code here
  Next item
End SyncLock
```

As effective as it is, **lock** isn't very friendly. When a thread acquires a lock in this manner, no other thread can access the collection. You should use **lock** only when performing an operation that must have exclusive access to the collection, and you should release the lock as soon as possible.

Synchronize is a much friendlier way to allow multiple threads access to a collection. It is a cooperative technique that requires each thread to follow the rules. Fortunately, those rules are very simple: rather than access the shared collection object, each thread obtains a reference to a synchronized collection object and performs all access through that reference. All it takes is a single call to the static **Synchronized** method, as shown below.

C#

```
Hashtable myHt = new Hashtable();
// code that adds items to the Hashtable
// obtain a synchronized wrapper around the Hashtable
Hashtable mySyncHt = Hashtable.Synchronize(myHt);
// all access to mySyncHt is synchronized
Console.WriteLine("myHt is {0}" ?
  myHt.IsSynchronized ? "synchronized" : "not synchronized");
Console.WriteLine("mySyncHt is {0}" ?
  mySyncHt.IsSynchronized ? "synchronized" : "not synchronized");
```

Visual Basic

```
Dim myHt as New Hashtable()
' code that adds items to the Hashtable
' obtain a synchronized wrapper around the Hashtable
Dim mySyncHt as Hashtable = Hashtable.Synchronize(myHt)
' all access to mySyncHt is synchronized
Dim msg As String
If myHt.IsSynchronized then
  msg = "synchronized"
```

```
Else
  msg = "not synchronized"
End If
Console.WriteLine("myHt is {0}", msg)
If mySyncHt.IsSynchronized then
  msg = "synchronized"
Else
  msg = "not synchronized"
End If
Console.WriteLine("mySyncHt is {0}", msg)
```

The **IsSynchronized** property returns **true** if the collection is synchronized.

It's worth repeating that using synchronized lists is *cooperative*. Nothing prevents a thread from accessing the list through the non-synchronized reference. But if you're careful to code the classes that use the list so all access it through the **Synchronized** reference, all will be fine.

In practice, you will use a combination of **Synchronize** and **lock** to provide thread-safe access to your collections. Most often, you'll use **Synchronize** during updates to prevent multiple threads from trying to modify the collection at the same time, and **lock** when you have a longer operation that requires exclusive access.

The other technique for providing thread-safe access involves creating a derived class and implementing your own **Synchronize** method. That is covered in the next section (next week), when we take a look at creating your own collection classes.

Type-Safe Collections

Last updated Jan 1, 2004.

The .NET generic collection types are very useful for all manner of data handling tasks. It's incredibly easy to create a collection, add and remove items, and enumerate or find items. The only real drawback is that, with the exception of language-specific array constructs, .NET arrays are inherently not type safe. If you're not careful, that can cause problems, especially when you work on a large program with many different programmers involved, or if you come back to a program a few months after you wrote it.

The problem is that collections store items of type **Object**. Since everything in a .NET program is an **Object**, you can put anything into a collection. You can also mix object types in a collection. It's perfectly legal, for example, to write:

```
ArrayList Cars = new ArrayList();
Car myCar = new Car();
Driver theDriver = new Driver();
Cars.Add(myCar);
Cars.Add(theDriver);
```

Now, normally you wouldn't want to mix object types in a collection. The whole *idea* of a collection is to group like objects. But since collections take **Object**s, the compiler can't prevent you from putting a **Driver** in the **Cars** collection. You'll run into trouble at runtime, though, when your enumeration code tries to cast a **Driver** to type **Car**. For example:

```
for (int i = 0; i < Cars.Count; i++)
{
  Car theCar = (Car)Cars[i];
  // do something with the Car
}
```

The cast is the giveaway here that things aren't entirely safe. The Framework will throw an invalid cast exception when this code tries to cast the **Driver** object that's in the collection to type **Car**.

The basic problem is that there's no built-in way to restrict the type of object that goes into a collection. If you want to do that, you have to create your own collection class.

The .NET Framework provides an abstract base class called **CollectionBase** (**MustInherit** in Visual Basic) that you can use as the basis for your custom collections. **CollectionBase** implements the **IList**, **IEnumerable**, and **ICollection** interfaces, and an inner **ArrayList** into which you can store items. The custom collection class becomes a type-safe wrapper around **ArrayList**. To create a custom collection in this manner, you have to implement the following:

From **ICollection**:

- **Count** property (implemented by **CollectionBase**)
- **IsSynchronized** property
- **SynchRoot** property
- **CopyTo** method

From **IList**:

- **IsFixedSize** property
- **IsReadOnly** property
- **Item** property
- **Add** method
- **Clear** method (implemented by **CollectionBase**)
- **Contains** method
- **IndexOf** method
- **Insert** method
- **Remove** method
- **RemoveAt** method (implemented by CollectionBase)

In short, you have to create type-safe versions of any method that takes a parameter of type **Object** or that returns an **Object**. This turns out to be reasonably straightforward, as you can see from the code below, which implements the **EmployeesCollection** type. The definition of the **Employee** class is shown after the collection code.

C#

```
class EmployeesCollection: CollectionBase
{
  // ICollection.IsSynchronized
  public virtual bool IsSynchronized
  {
    get { return false; }
  }

  // ICollection.SyncRoot
  public virtual EmployeesCollection SyncRoot
  {
    get { return this; }
  }

  // ICollection.CopyTo
  public virtual void CopyTo (Employee[] emps, int index)
  {
    InnerList.CopyTo (emps, index);
  }

  // IList.IsFixedSize property
  public virtual bool IsFixedSize
  {
    get { return InnerList.IsFixedSize; }
  }

  // IList.IsReadOnly property
  public virtual bool IsReadOnly
  {
    get { return InnerList.IsReadOnly; }
  }

  // IList.Item property
  public virtual Employee this[int index]
  {
    get { return (Employee)(InnerList[index]); }
    set { InnerList[index] = value; }
  }

  // IList.Add
  public virtual int Add (Employee emp)
  {
    return InnerList.Add (emp);
  }

  // IList.Contains
  public virtual bool Contains (Employee emp)
  {
```

```
    return InnerList.Contains (emp);
  }

  // IList.IndexOf
  public virtual int IndexOf (Employee emp)
  {
    return InnerList.IndexOf (emp);
  }

  // IList.Insert
  public virtual void Insert (int index, Employee emp)
  {
    InnerList.Insert (index, emp);
  }

  // IList.Remove
  public virtual void Remove (Employee emp)
  {
    InnerList.Remove (emp);
  }
}
```

Visual Basic

```
Public Class EmployeesCollection
  Inherits CollectionBase

  ' ICollection.IsSynchronized
  Public Overridable ReadOnly Property IsSynchronized() As Boolean
    Get
      Return False
    End Get
  End Property

  ' ICollection.SyncRoot
  Public Overridable ReadOnly Property SyncRoot() As
➥EmployeesCollection
    Get
      Return Me
    End Get
  End Property

  ' ICollection.CopyTo
  Public Overridable Sub CopyTo(ByVal emps() As Employee, ByVal index
➥As Integer)
    InnerList.CopyTo(emps, index)
  End Sub

  ' IList.IsFixedSize property
  Public Overridable ReadOnly Property IsFixedSize() As Boolean
    Get
      Return InnerList.IsFixedSize
    End Get
  End Property

  ' IList.IsReadOnly property
```

```
  Public Overridable ReadOnly Property IsReadOnly() As Boolean
    Get
      Return InnerList.IsReadOnly
    End Get
  End Property

  ' IList.Item property
  Default Public Overridable Property Item(ByVal index As Integer)
    Get
      Return DirectCast(InnerList(index), Employee)
    End Get
    Set(ByVal Value)
      InnerList(index) = Value
    End Set
  End Property

  ' IList.Add
  Public Overridable Function Add(ByVal emp As Employee) As Integer
    Return InnerList.Add(emp)
  End Function

  ' IList.Contains
  Public Overridable Function Contains(ByVal emp As Employee) As
➥Boolean
    Return InnerList.Contains(emp)
  End Function

  ' IList.IndexOf
  Public Overridable Function IndexOf(ByVal emp As Employee) As
➥Integer
    Return InnerList.IndexOf(emp)
  End Function

  ' IList.Insert
  Public Overridable Sub Insert(ByVal index As Integer, ByVal emp As
➥Employee)
    InnerList.Insert(index, emp)
  End Sub

  ' IList.Remove
  Public Overridable Sub Remove(ByVal emp As Employee)
    InnerList.Remove(emp)
  End Sub
End Class
```

The methods and properties are **virtual** (**Overridable** in Visual Basic) so that you can make collections for derived classes. No heavy lifting required, just a lot of typing.

That "lot of typing" can get tedious, though, if you're creating more than one or two custom collection classes. With a little study, you can create a program that uses the Code DOM to generate the collection classes for you, as described in the documentation article, Generating and Compiling Source Code Dynamically in Multiple Languages, and in the MSDN Magazine article, Bring the Power of Templates to Your .NET Applications with the CodeDOM Namespace, by Adam J. Steinert.

Most of the typing associated with creating type-safe collection classes will be eliminated with the next version of the .NET Framework. That version will implement *generic types*, which are similar to C++ templates and allow you to create type-safe collection classes (and other things) quickly and easily with a minimal amount of typing.

The **Employee** class on which the code above depends is shown here:

C#

```
class Employee
{
  private string firstName;
  private string lastName;

  public Employee (string fName, string lName)
  {
    firstName = fName;
    lastName = lName;
  }

  public string FirstName
  {
    get { return firstName; }
    set { firstName = value; }
  }

  public string LastName
  {
    get { return lastName; }
    set { lastName = value; }
  }
}
```

Visual Basic

```
Public Class Employee
  Private fName As String
  Private lName As String

  Public Sub New(ByVal first As String, ByVal last As String)
    fName = first
    lName = last
  End Sub

  Public Property FirstName() As String
    Get
      Return fName
    End Get
    Set(ByVal Value As String)
      fName = Value
    End Set
  End Property

  Public Property LastName() As String
```

```
    Get
      Return lName
    End Get
    Set(ByVal Value As String)
      lName = Value
    End Set
  End Property
End Class
```

You use this collection type just as you would any other, except that you can only put **Employee** objects (or objects that are derived from **Employee**) into it. If you try to put a **Car** object into the **EmployeesCollection**, the compiler will issue an error message.

Synchronization

Last updated Jan 1, 2004.

The type-safe collection code shown above implements the **SyncRoot** property, which allows the collection to be used in a thread-safe manner. As discussed in the previous section, though, **SyncRoot** isn't the friendliest interface ever invented. It requires that clients explicitly **lock** (**SyncLock**) the collection at each call, and release the lock when they're finished using the collection. This can work, but it's all too easy for a client to hold the lock for too long, or to forget to obtain the lock in the first place. The **Synchronized** method is a much nicer interface, but requires a bit more work to implement.

The idea behind implementing a **Synchronized** method is to provide a wrapper that performs the required **lock** operations as required. This ensures that the client won't forget to obtain the locks and that locks are held only as long as required. What you do is create a derived class that wraps the critical operations, and a **static** (**Shared** in Visual Basic) method in the collection class that returns a reference to the *real* collection. The code below shows how it's done in the **EmployeesCollection** class, although it doesn't implement all of the methods.

C#

```
// EmployeesCollection.Synchronized
// returns a reference to a thread-safe collection wrapper
public static EmployeesCollection Synchronized(
  EmployeesCollection emps)
{
  return new SynchronizedEmployeesCollection(emps);
}
// SynchronizedEmployeesCollection is a thread-safe wrapper
// around EmployeesCollection
// Not all methods are implemented.
class SynchronizedEmployeesCollection: EmployeesCollection
{
  EmployeesCollection emps;
```

```
  public SynchronizedEmployeesCollection(EmployeesCollection ec)
  {
    emps = ec;
  }

  public override int Add (Employee emp)
  {
    lock (emps.SyncRoot)
    {
      return emps.Add(emp);
    }
  }
}
```

Visual Basic

```
' EmployeesCollection.Synchronized
' returns a reference to a thread-safe collection wrapper
Public Shared Function Synchronized( _
ByVal emps As EmployeesCollection) As EmployeesCollection
  Return New SynchronizedEmployeesCollection(emps)
End Function

' SynchronizedEmployeesCollection is a thread-safe wrapper
' around EmployeesCollection
' Not all methods are implemented.
Public Class SynchronizedEmployeesCollection
  Inherits EmployeesCollection
  Dim emps As EmployeesCollection

  Public Sub New(ByVal ec As EmployeesCollection)
    emps = ec
  End Sub

  Public Overrides Function Add(ByVal emp As Employee) As Integer
    SyncLock emps.SyncRoot
      Return emps.Add(emp)
    End SyncLock
  End Function
End Class
```

You would have to provide overrides for all of the **EmployeesCollection** methods. Then, your main program should keep the original **EmployeesCollection** instance private, and publish only the reference to the synchronized collection: the **SynchronizedEmployeesCollection**. This ensures that all threads that access the collection do it through the thread-safe wrapper.

Difference Between String and StringBuilder

String Class	StringBuilder
Once the string object is created, its length	Even after object is created, it can be able

and content cannot be modified.	to modify length and content.
Slower	Faster

Limitation of Arrays

- The **size of an array is always fixed** and must be defined at the time of instantiation of an array.
- Secondly, an **array can only contain objects of the same data type**, which we need to define at the time of its instantiation.

ArrayList Concept in .Net

```
Provides a collection similar to an array, but that grows dynamically
as
the number of elements change.
```

Example

```
static void Main()
{
    ArrayList list = new ArrayList();
    list.Add(11);
    list.Add(22);
    list.Add(33);
    foreach(int num in list)
    {
        Console.WriteLine(num);
    }
}
```

Output

```
11
22
33
```

Stack Concept in .Net

```
A collection that works on the Last In First Out (LIFO) principle,
i.e., the last item inserted is the first item removed from the
collection.
```

Push - To add element and
Pop – To Remove element

Example

```
using System;
using System.Collections;

class Test
{
    static void Main()
    {
        Stack stack = new Stack();
```

```
        stack.Push(2);
        stack.Push(4);
        stack.Push(6);

        while(stack.Count != 0)
        {
            Console.WriteLine(stack.Pop());
        }
    }
}
```

Output
6
4
2

Queue Concept in .Net

A collection that works on the First In First Out (FIFO) principle,
i.e.,
the first item inserted is the first item removed from the collection.
Enqueue - To add element and Dequeue – To Remove element
Example:

```
static void Main()
{
    Queue queue = new Queue();
    queue.Enqueue(2);
    queue.Enqueue(4);
    queue.Enqueue(6);

    while(queue.Count != 0)
    {
        Console.WriteLine(queue.Dequeue());
    }
}
```

Output
2
4
6

Dictionaries Concept in .Net

Dictionaries are a kind of collection that store items in a key-value pair fashion.

System.Collections namespace

Hashtable Concept in .Net

Provides a collection of key-value pairs that are organized
based on the hash code of the key.

Example:

```
static void Main()
{
    Hashtable ht =  new Hashtable(20);
    ht.Add("ht01", "DotNetGuts");
    ht.Add("ht02", "EasyTutor.2ya.com");
    ht.Add("ht03", "DailyFreeCode.com");

    Console.WriteLine("Printing Keys...");
    foreach(string key in ht.Keys)
    {
        Console.WriteLine(key);
    }

    Console.WriteLine("\nPrinting Values...");
    foreach(string Value in ht.Values)
    {
        Console.WriteLine(Value);
    }

    Console.WriteLine("Size of Hashtable is {0}", ht.Count);

    Console.WriteLine(ht.ContainsKey("ht01"));
    Console.WriteLine(ht.ContainsValue("DailyFreeCode.com"));

    Console.WriteLine("\nRemoving element with key = ht02");
    ht.Remove("ht02");

    Console.WriteLine("Size of Hashtable is {0}", ht.Count);

}
```

Output

Printing Keys...
ht01
ht02
ht03

Printing Values...
DotNetGuts
EasyTutor.2ya.com
DailyFreeCode.com

Size of Hashtable is 3
True
True

Removing element with key = ht02
Size of Hashtable is 2