Exam Manual

LearnSmart

Microsoft .NET Framework 2.0 (70-526)

Windows-based Application Development

If you are an experienced developer seeking certification, this LearnSmart exam manual can help you prepare for Microsoft Exam 70-526, which measures your ability to develop and implement Windows-based applications with the Microsoft .NET Framework 2.0. Topics covered in this manual include:

- Integrating Data in a Windows Forms Application
- Enhancing Usability
- Developing Windows Forms Controls
- Configuring and Deploying Applications
- And more!

Jumpstart your way to the top of the ladder by purchasing this exam manual today!

Microsoft .NET Framework 2.0 Windows-based Application Development (70-526) Learn SmartExam Manual

Product ID: 010878
Production Date: July 22, 2011

Warning and Disclaimer

Every effort has been made to make this document as complete and as accurate as possible, but no warranty or fitness is implied. The publisher and authors assume no responsibility for errors or omissions. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this document.

Volume, Corporate, and Educational Sales

Favorable discounts are offered on all products when ordered in quantity. For more information, please contact us directly:

1-800-418-6789
solutions@learnsmartsystems.com

International Contact Information

International: +1 (813) 769-0920

United Kingdom: (0) 20 8816 8036

Table of Contents

Abstract

This Exam Manual is designed to accomplish two goals. First, it is prepared to help familiarize you with the inner workings of the new .NET Framework 2.0 and second, it is designed to help you in your preparation to pass the Microsoft 70-526: Windows Based Client Development examination, which is considered to be one of the most difficult Microsoft certification paths in the industry.

While studying this Exam Manual, you should continually ask yourself whether you *Know and Understand* the material presented, or if you are simply just familiar with it. In order to pass the examination, you will need to feel as if you know the .NET Framework backwards and forwards. If not, you chance the risk of having a very minor part of the Framework throw off your work as a whole. By passing both the 70-536 and 70-526, you will be certified as an official Microsoft Certified Technology Specialist on Windows development.

Tips

This exam requires a thorough understanding of the .NET Framework as well as the new capabilities specifically provided by the .NET 2.0 Framework. Depending on the programming language you select, you will need to be familiar with the Visual Basic 2005, Visual C# 2005, or Visual C++ 2005. You will not be required to write and compile code, but you should be familiar with syntax and structure of the language selected.

The .NET Framework 2.0 does not introduce a new development framework or paradigm, implementation model, or scripting model from earlier releases of .NET Framework 1.0 or .NET Framework 1.1. However, because it is much more than an incremental release, it introduces significant enhancements to the .NET Framework. This manual does not cover each technical item in detail, but it provides a targeted overview of the material in this exam and prepares you to understand the topics outlined in the Technology Specialist (TS) Exam 70-526: TS: Microsoft .NET Framework 2.0—Windows Based Client Development exam.

What to Know

The best way to prepare for this exam is to read all the material you can find related the .NET and .NET Framework 2.0. Also, you should try to get as much hands on experience with the .NET framework as possible. This is accomplished in three ways: by Coding, Reading, and Studying. Microsoft provides a wonderful web resource called the MSDN library that contains sample code, documentation, and other materials that you can use to further your knowledge of the .NET Framework.

It's a good idea to also participate in several large projects. Whether you do these projects at your office or you simulate them on your own, it's wise to involve yourself in the practical coding of large scale multi-threaded .NET intensive apps that use a wide variety of new extensions to the .NET Framework you will find in this Exam Manual. Also, you should check back at www.PrepLogic.com to see the newly upcoming leading practice exam, which will be available for an extremely low price.

Lastly, it's highly recommended that you familiarized yourself with the objectives of the exam at:
http://www.microsoft.com/learning/exams/70-526.asp

1.0 Creating a UI for a Windows Forms Application by Using Standard Controls

1.1 Adding and Configuring a Windows Form

A Windows form is the basic foundational element in a .NET application. All Windows forms can be highly customized and adapted to the user interface requirements. Controls are added to give forms functionality. Controls can exist in either their default form or they can be highly customized by the developer. The combination of Windows forms and controls allow for an almost unlimited range of applications.

Adding a Windows Form to a Project at Design Time

Once a project is created (unless the application is a console application), a form or multiple forms are added to the project to serve as the basis for application functionality. The default form that is added to the project when it is created is called Form1.

There are two general methods for adding forms to a project:

1. Add a form to a project at design time.
2. Add a form to an application at run time.

The standard method for adding a form to a project is to use the Visual Studio IDE:

1. Select the Project Menu.
2. Select Add Windows Form.
3. Select Windows Form from the dialog box.
4. Create a name for the form.
5. Click Add.

The new Windows form is now ready for configuration. Any number of new forms can be added to the project.

The standard method for adding a form at runtime is through the code itself, which creates an instance of the form:

Visual Basic:

```
Dim appForm as FormX
appForm = new FormX()
appForm.Show()
```

C#:

```
FormX appForm;
appForm = new FormX();
appForm.Show();
```

The **Form** class manages the Windows forms in a project. It can be used to create various Windows forms, such as standard, borderless, and floating forms. The **Form** class manages the size, appearance, color, and management features of Windows forms within an application. It also allows for the response to various events on the form.

Form Class

Namespace: System.Windows.Forms

Assembly: System.Windows.Forms (in system.windows.forms.dll)

There are literally dozens of methods that are members of the **Form** class, which is one of the larger classes in the .NET Framework.

Configuring a Windows Form to Control Functionality

The **Form** class allows for the easy configuration of form control functionality. Much of the functionality is contained visually within the Properties window, which has controls for configuring accessibility, appearance, behavior, data, design, focus, layout, and style. You can configure these areas through dropdown boxes or by manually typing in a setting within the prescribed field.

1.2 Managing Control Layout on a Windows Form

Grouping and Arranging Controls

The use of controls is vital to the successful programming of Windows forms. A specialized group of controls, called container controls, act as a container for other controls on the form, grouping them logically for ease of programming and usability. Containers can be enabled and disabled. When a container control is disabled, none of the controls that are a part of the container will be active.

The following are primary container controls that are found in the .NET 2.0 Framework:

Panel

The Panel container control forms a subsection of a form that allows for various other controls to be placed on it. A Panel can be borderless, making it appear to be indistinguishable from the parent form, or it can be assigned a border through the **BorderStyle** property.

The **BorderStyle** property has three important settings:

Setting	Description
None	No border. Panel seems integrated with parent form.
FixedSingle	A single edge is present around the border.
Fixed3D	A border with a three-dimensional appearance is present.

A Panel also is capable of scrolling. The **AutoScroll** property, when set to **True**, provides a scroll bar; when set to **False**, a scroll bar is not present and only the visible controls within the Panel are accessible.

GroupBox

A GroupBox is like a Panel in that it is a subdivision of the parent form; however, it does not have scrolling capabilities and it does not have border settings or layout functionality like a Panel.

The main property for a GroupBox is a caption. The **Text** property controls the captioning for a GroupBox and has two states: either text is added or an empty string exists, denoting no caption.

GroupBoxes are commonly used to group a set of similar controls, such as RadioButtons or CheckBoxes.

TabControl

The TabControl container control simply allows the programmer to group controls on tabs so that the user can tab through various controls to perform actions on the form.

The main property of the TabControl container control is the **TabPage** property, which allows individual configuration of the properties of each **TabPage**.

TabPage has both of the key properties of the Panel (**Border Style** and **AutoScroll**) and the **Text** property of the GroupBox.

The TabControl container has four important properties:

Property	Description
Appearance	Indicates whether the tabs are displayed as buttons or regular tabs
Alignment	Determines whether tabs are aligned to the top, bottom, left, or right of the control
Multiline	Indicates whether more than one row of tabs is allowed
TabPages	A collection of TabPage controls to be used by the TabControl property

FlowLayoutPanel

The FlowLayoutPanel container control is quite similar to the Panel container control. Its main difference is that it dynamically allows for the repositioning of its enclosed controls as it is resized, both at design and run time. Like the Panel control, it contains the **AutoScroll** property.

By default the direction of control is from left to right. This can be changed through the **FlowDirection** property, which has four values:

1. **LeftToRigh**t
2. **RightToLeft**
3. **TopDown**
4. **ButtomUp**

The **WrapContents** property allows the flow to be wrapped to the next column or row. If set to **True**, the default setting will wrap automatically. If set to **False**, the controls will not wrap to the next column or row and will be clipped.

The **SetFlowBreak** method allows you to set the flow break on a control.

The **GetFlowBreak** method, paired with the **SetFlowBreak** method, returns a **True** or **False** value when a flow break has been set.

TableLayoutPanel

The TableLayoutPanel is a container control that allows you to establish cells in which controls can reside. Each cell usually hosts a single control, although multiple controls are possible in advanced designs.

As with the Panel container control, the TableLayoutPanel contains the **AutoScroll** property.

The **CellBorderStyle** property controls the appearance of the cells within the table and has the following five properties:

1. **None**
2. **Single**
3. **Inset** or **InsetDouble**
4. **Outset** or **OutsetDouble**
5. **OutsetPartial**

Other main properties of the TableLayoutPanel include:

Property	Description
ColumnCount	The number of columns in the table.
Columns	A collection of the column styles of the table. In Visual Studio, this launches the Columns and Rows Styles editor.
ColumnStyles	A collection of **ColumnStyles** that represent the look and feel of a column.
GrowStyle	Allows for the growing of the TableLayoutPanel through the following three settings: 1. **AddColumns** 2. **AddRows** 3. **FixedSize**
Rows	A collection of the row styles of the table. In Visual Studio, this launches the Columns and Rows Styles editor.
RowStyles	A collection of **Row Styles** that represent the look and feel of a row.

The TableLayoutPanel calls one primary method, the **Controls.Add** method of **System.Windows.Controls**, to add controls to the current TableLayoutPanel.

Using the SplitContainer Control to Create Dynamic Container Areas

The SplitContainer control allows you to create a panel that is divided into two panels named Panel1 and Panel2. Each panel is set by SplitterPanel controls, which are similar to Panel controls. The SplitContainer has a **BorderStyle** property like the other panels.

The **FixedPanel** property allows a panel to be fixed, and has three possible settings:

1. **Panel1**
2. **Panel2**
3. **None**

The **SplitContainer** control has several unique properties:

Property	Description
IsSplitterFixed	Determines whether the splitter is fixed or can be adjusted.
Orientation	Sets the horizontal or vertical orientation of the splitter.
Panel1/Panel2	Exposes the properties of SplitterPanel 1 or 2.
Panel1(2)Collapsed	If collapsed, the setting is **True**; otherwise, it is **False**.
Panel1(2)MinSize	Sets the minimum size of the panels.
SplitterDistance	Sets the distance of the splitter from the top or left edge.
SplitterWidth	Sets the width of the splitter.

1.3 Adding and Configuring a Windows Form Control

Using the IDE to Add a Control to a Windows Form or Other Container Control of a Project at Design Time

There are several ways to add a control to a form at design time. The most popular ways are to access the control in the Toolbox and do one of the following:

1. Double click the control.
2. Drag the control to the form.
3. Select the control and then double click the form.
4. Select a control and then draw it on the form via the mouse.

Adding Controls to a Windows Form at Run Time

To add a control in a Windows form at runtime requires the manual development of code. The following is a general procedure to add a control at runtime:

1. Instantiate the new control via code.
2. Set the properties of the control in the code.
3. Add the new control to the form's Controls collection.

Modifying Control Properties

Control properties can be modified in many ways in the .NET Framework. The four most common ways consist of the following:

1. The Designer
2. The Properties window
3. SmartTags
4. The Document Outline window

Configuring Controls on a Windows Form at Run Time to Ensure that the UI Complies with Best Practices

The end user must be kept in mind when configuring controls on a Windows form. If users have difficulty understanding the application, their productivity will decrease and their frustration will increase. Therefore, you must keep a couple considerations in mind when designing forms.

First, you should facilitate optimal human-computer interaction, which allows the user to interact effectively and efficiently with the application. Second, keep it simple, which means that the form should contain only the essential controls that are needed to accomplish the given task. This includes strategically placed controls and using controls with a layout that makes sense to users.

Consistency is another characteristic that facilitates user productivity. Without a consistent form throughout the application, users will get frustrated and may make mistakes when using the application.

Creating and Configuring Command Controls

The purpose of command controls is to execute a task or to continue with an operation of the application. One of the primary command controls is the Button control. When a user clicks on a Button control, a series of code is executed.

The **Button_Click** method is the primary method and the **Button.Click** event handler is the primary event handler.

You can use **Mouse** events instead of **Button** events.

The special cases of Accept or Cancel buttons are created through the **DialogResult** property of the Button control. In the **DialogResult** property in the Properties window of the button, set the property to **OK** for an Accept Button or to **Cancel** for a Cancel Button.

Creating and Configuring Text Edit and Display Controls

Labels are used to display read-only information on a Windows form. They can also be used as shortcut keys for other controls. The Label control is used to configure Labels on the Windows form. You set the Label's properties in the Properties window.

LinkLabels are controls that create Web links, which open a Web page from the application. LinkLabels have several properties:

Property	Description
ActiveLinkColor	Sets the active link color
LinkArea	Sets the portion of the label that acts as a link
LinkBehavior	The prescribed behavior of the link
LinkColor	Sets the link's color
LinkVisited	Shows whether the link was visited
VisitedLinkColor	Sets the visited link's color

Using List-Based Controls

List-based controls organize data and present it in a clear manner to the user. The primary list-based controls are the ListBox, the ComboBox, and the CheckedListBox. Each of these controls contains an **Items** collection that provides organizational functionality to the specified control.

Items collections are collections of objects that are usually in the form of strings, although a string is not required.

ListBox

The ListBox is the most basic of the list-based controls. It displays a list of items in a simple, concise manner, and allows the selection of one or more items.

ComboBox

The ComboBox control contains the same basic functionality as the ListBox with the added functionality of permitting an addition to the list through the manual typing of a string entry.
A ComboBox can allow for the following displays:

1. A List display
2. A Drop Down display

CheckedListBox

A CheckedListBox presents a list of checkable boxes to the user; the user can check one or more of the boxes.

Configuring a WebBrowser Control

The WebBrowser control allows for the loading and displaying of Web pages and also provides critical functionality for Web navigation. The WebBrowser control consists of literally dozens of methods and properties to facilitate access to Web pages.

The main method of the WebBrowser control is the **Navigate** method. This simple method accepts a string for the URL and loads the string into the WebBrowser control to load the Web page. The following code demonstrates the use of the **Navigate** method:

Visual Basic:

```
WebBrowserAlpha.Navigate("www.acm.org")
```

C#:

```
webBrowserAlpha.Navigate("www.acm.org");
```

Upon navigating to the specified site, the WebBrowser control raises the **DocumentCompleted** event.

If code needs to execute after the Web page is loaded, the **DocumentCompleted** event should be handled.

Adding and Configuring Date Setting Controls

The DateTimePicker control allows users to set a date, a time, or both through a user interface on the form. A user can use a drop-down box that displays a calendar to select a day, or they type in the time in a text area.

The DateTimePicker control has several properties:

Property	Description
CustomFormat	The custom display format used when the **Format** property is set to **Custom**
Format	Sets the date and time display format
MaxDate	The maximum date the control will accept
MinDate	The minimum date the control will accept
Value	The current **DateTime** value of the control

Displaying Images by Using Windows Forms Controls

Displaying images adds a certain aesthetic quality to an application, as well as functionality. The main control to facilitate the displaying of images on a Windows form is the PictureBox control.

The PictureBox control can use images of the following six formats:

1. .bmp
2. .jpg/.jpeg
3. .gif
4. .png
5. .wmf (Metafiles)
6. Icons

The following are important properties of the PictureBox control:

Property	Description
ErrorImage	The alternate image to display if the specified image fails to load
Image	The image to be displayed
ImageLocation	The address to the image
InitialImage	The image to be displayed while another image is loading
SizeMode	Determines how the control will handle placement and sizing of the image

Images in a Windows form can also be organized. This may be necessary because multiple controls in the application may require the same image.

The ImageList component creates a repository of images for use by the application. It has several properties:

Property	Description
ColorDepth	Sets the number of colors to use when rendering the images
Images	The organized collection of images
ImageSize	The size of each image in the ImageList

The images from the ImageList can also be accessed at run time through the **Images** collection by using the following code:

Visual Basic:

```
PictureBoxAlpha.Image = ImageListAlpha.Images(1)
```

C#:

```
pictureBoxAlpha.Image = imageListAlpha.Images[0];
```

Note: Images can be added to Buttons, CheckBoxes, RadioButtons, and other controls through the ImageList properties. These images can be provided by the ImageList repository.

Configuring the NotifyIcon Component

The NotifyIcon component represents an icon in the system tray. The NotifyIcon component most often is used with applications that run in the background. In many cases, this component is also used to show balloon tips to the user:

Visual Basic:

```
NotifyIconAlpha.ShowBalloonTip(15)
```

C#:

```
notifyIconAlpha.ShowBalloonTip(15)
```

In this case, the number 15 indicates the number of seconds the balloon tip will be displayed on the screen.

The NotifyIcon component has several properties:

Property	Description
BalloonTipIcon	Sets the icon that will be displayed. The property can be set to display: 1. None 2. Info 3. Warning 4. Error
BalloonTipText	Sets the text that will be displayed.
BalloonTipTitle	Sets the balloon tip title.
ContextMenuStrip	Sets the **ContextMenuStrip** associated with the particular instance of NotifyIcon.
Icon	The displayed icon on the system tray.
Text	The text displayed on the system tray when the mouse hovers over it.
Visible	The status of the visibility of the icon on the system tray.

Creating Access Keys for Windows Forms Controls

When creating an access key, such as a Label control, you'll need to follow a simple process to activate it:

1. Drag the control, such as a Label control, onto the form in a location close to the control with which the access key will correspond.
2. Type in the Text of the control.
3. Place an ampersand in front of the letter or character that will act as the key.
4. Set the **Use Mnemonic** property to **True**.
5. Set the **TabIndex** to one less than the **TabIndex** property of the control being defined as an access key.

Once these steps are completed, the access key will be fully operational.

1.4 Creating and Configuring Menus

Creating and Configuring a MenuStrip Component

MenuStrip controls are designed for the efficient display of ToolStripMenuItems. As such, they are derived directly from the ToolStrip component, but are designed primarily to host the ToolStripMenuItems, which can consist of:

1. Text
2. Images
3. Executable code
4. A combination of the above

The creation of a MenuStrip at design time is quite simple: it is dragged onto the form from the Toolbox. Once on the form, an interface appears that allow you to create any number of menu items.

The configuring of menu items is quite simple as well. The Properties window contains various properties for each type of control placed on the form; these can be changed through manual typing or through drop-down boxes.

Several menu items can have their properties changed at once to ensure the uniformity of a particular form property. This is done by holding the Ctrl key while simultaneously clicking on the menu items that are to have their properties changed.

Changing the Displayed Menu Structure Programatically

Using code to change the Menu structure is quite simple, and allows for the instantaneous swapping of menus on the user interface through the use of the **Controls** collection of **System.Windows.Controls**. The following demonstrates the use of the **Controls Add** and **Remove** methods:

Visual Basic:

```
Me.Controls.Remove(MenuStripB)
Me.Controls.Add(MenuStripC)
```

C#:

```
this.Controls.Remove(MenuStripB);
this.Controls.Add(MenuStripC);
```

Creating and Configuring the ContextMenuStrip Component

The ContextMenuStrip component allows for the creation of context menus, which are the shortcut menus that appear when an object is right-clicked. The ContextMenuStrip and the MenuStrip components are quite similar, with the main differences being that the ContextMenuStrip does not have a top-level menu and is not visible unless an object is right-clicked.

ContextMenuStrip controls can be added by using the **ContextMenuStrip.Item.Add** method and removed by using the **ContextMenuStrip.Item.Remove** method.

Once added, a ContextMenuStrip must be associated with a particular control. This association is set, along with other properties, in the Properties window of the specific control.

1.5 Creating Event Handlers for Windows Forms and Controls

Using the Windows Forms Designer to Create Event Handlers

Events are messages that indicate that something is happening within an application. The instant an event is raised, objects within the application are given a chance to respond to the event through the use of their corresponding event handler, which in turn executes a method or procedure in response to the event.

Each class or control has its own corresponding events that relate to its functionality. On Windows forms, each form or control within the form can raise an event based on actions such as user input to the form. Primary events are created by the user through the mouse or keyboard.

Events that are raised by controls on a form contain two parameters:

1. Parameters carrying an object reference to the raising control
2. Parameters that carry event arguments

Event handlers are created through the Properties window of a control. The "Lightening Bolt" icon in the Properties window facilitates the display of event handlers.

Managing Mouse and Keyboard Events and Programming a Windows Forms Application to Recognize Modifier Keys

Most of the events that are handled through user input occur through the use of the mouse and keyboard.

Mouse events are standard events that occur in the same manner with most applications.

Mouse activities include:

1. Clicks or double clicks
2. Entering/leaving the control
3. Moving the control
4. Hovering over the control
5. Scrolling by using the mouse wheel

The most common events are the **MouseClick** and **MouseDoubleClick** events. These are simple events that return Object references to the control that raised the event.

Movement events include the following actions with a control:

1. **MouseEnter**
2. **MouseHover**
3. **MouseLeave**

Once again, these are simple events that return Object references to the control that raised the event. There are several events that pass **MouseEventArgs**. These include:

1. **MouseDown**: Mouse button pressed over a control.
2. **MouseUp**: Mouse button released over a control.
3. **MouseMove**: Mouse moved over a control.
4. **MouseWheel**: Mouse wheel moved.

The **MouseEventArgs** that are passed are standard and consist of the following properties:

Property	Description
Button	Gets which mouse button was pressed
Clicks	The number of times the button was pressed
Delta	The number of clicks the mouse wheel was rotated
Location	The current mouse location
X	The *X* coordinate location
Y	The *Y* coordinate location

Keyboard input controls can raise three events:

1. **KeyDown**
2. **KeyUp**
3. **KeyPress**

An instance of **KeyEventArgs** is raised upon the keyboard events. The basic three properties are based on the standard modifier keys as follows:

1. **Alt**: The Alt key is pressed.
2. **Control**: The Ctrl key is pressed.
3. **Shift**: The Shift key is pressed.

Other properties of **KeyEventArgs** are as follows:

Property	Description
Handled	Indicates whether an event was handled
KeyCode	Returns an enum value indicating the pressed key
KeyData	Returns the value of the pressed key in conjunction with the corresponding Ctrl, Alt, or Shift key
KeyValue	An integer representing the **KeyData** property
Modifiers	Gets or sets modifier flags indicating the combination of the corresponding Ctrl, Alt, or Shift keys
SupressKeyPress	Sets a value indicating whether the key event should be passed on to the control

Using the Windows Forms Designer to Create Default Event Handlers

A default event handler is a method that handles a given event for a control. It is created through the Properties window by clicking on the "Lightening Bolt" icon and then using the Code Editor to add manual code to execute when the event is raised. In good practice, a descriptive name is assigned to the default event handler.

Connecting Multiple Events to a Single Event Handler

Multiple events can be assigned to the same event handler. The only requirement is that the signature of the method must match the signature of the event. This can be set up in the same manner that single events are created, through the Properties window.

Using the Code Editor to Override Methods Defined in the Base Class

Overriding a method that has been defined in a base class prevents that method from gaining functionality. This creates a new implementation of the method to account for different functionality required in the current instance of the class.

The implementation of an override is simple. In the program code type the following after the class declaration but before the method declaration:

Visual Basic:

```
// Class Declaration
Overrides
//Method Declaration
```

An example of this format would be:

```
Public Class FormA
Overrides
AutoSize() As Boolean
```

C#:

```
// Class Declaration
override
// Method Declaration
```

An example of this format would be:

```
Public partial class FormA : Form {
override
public AutoSize{get; set; };
}
```

Once this code is injected, the method is overridden.

2.0 Integrating Data in a Windows Forms Application

2.1 Implementing Data-Bound Controls

The integration of data in an application is absolutely essential for strategic data management. .NET 2.0 Framework's data-bound controls make the manipulation of data easy and manageable, and allow for advanced data functionality and usage. Data-bound controls can be set through code, smart tags, and menus in the visual workspace.

Using the DataGridView Control to Display and Update the Tabular Data Contained in a Data Source

The DataGridView has many purposes, but is commonly used to show the DataTable contents in a DataSet. This is achieved through a simple set of code. To show the DataTable in a DataView perform the following:

1. Set the **DataSource** property of DataGridView to the specific DataSet.
2. Set the **DataMember** property to the name of the DataTable.

The code is straightforward:

Visual Basic:

```
DataGridViewX.DataSource = FinanceDataSet
DataGridViewX.DataMember = "NewAccounts"
```

C#:

```
dataGridViewX.DataSource = financeDataSet;
dataGridViewX.DataMember = "NewAccounts";
```

Note: Smart tags can also be used on a DataGridView control by selecting the appropriate control when a smart tag is displayed.

Using a Simple Data-Bound Control to Display a Single Data Element on a Windows Form

The process of data binding is quite simple. Basic data binding places a single data element in a control. Multiple sources of data can also be bound to a single control, such as in a drop-down menu. Both single and multiple data sources are bound in the same manner.

Single data binding involves matching a single datum with a control.

The data can be bound to the control as in the following example, where a data element is bound to a text box:

Visual Basic:

```
TextBoxX.DataBindings.Add("Text", newAccountsBindingSource, "New Accounts")
```

C#:

```
textBoxX.DataBindings.Add("Text", newAccountsBindingSource, "New Accounts");
```

This code binds the NewAccounts column from the DataTable to the required text box.

Implementing Complex Data Binding to Integrate Data from Multiple Sources

Integrating data from multiple sources follows the same basic concepts as does binding a single data source. Two basic properties are used with multiple data sources:

1. The **DataSource** property
2. The **DataMember** property

Two basic objects will serve as the object of the **DataSource** property:

1. A **BindingSource** object
2. A **DataSet** object

The **DataMember** property will usually be the corresponding table of the database.

As with single data binding, the binding source is set up and then the data is bound to the controls:

Visual Basic:

```
Dim newAccountsBindingSource  As New BindingSource(FinanceDataSet,
"NewAccounts")
DataGridViewX.DataSource = newAccountsBindingSource
```

C#:

```
BindingSource newAccountsBindingSource = new BindingSource(financeDataSet,
"NewAccounts");
dataGridViewX.DataSource = newAccountsBindingSource;
```

Now, the DataGridView is bound to the table:

Visual Basic:

```
DataGridViewX.DataSource = FinanceDataSet
DataGridViewX.DataMember = "NewAccounts"
```

C#:

```
dataGridViewX.DataSource = financeDataSet;
dataGridViewX.DataMember = "NewAccounts";
```

The complex data is now bound appropriately to the corresponding table.

Navigating Forward and Backward through Records in a DataSet in Windows Forms

The BindingNavigator component is used to navigate forward and backward in a data source.

Through the **BindingNavigator.BindingSource** property, the following methods can be invoked:

1. **MoveNext**
2. **MovePrevious**

Defining a Data Source by Using a DataConnector Component

The purpose of using a DataConnector instead of a DataSet is to have the ability to redirect the application to an alternate data source without the burden of having to redirect all of the current data binding code, as seen in the previous section.

This is accomplished through the BindingSource component.

The following code sets up the binding source using the BindingSource component:

Visual Basic:

```
newAccountsBindingSource = New BindingSource(FinanceDataSet,"NewAccounts")
```

C#:

```
newAccountsBindingSource = new BindingSource(financeDataSet,"NewAccounts");
```

2.2 Managing Connections and Transactions

Configuring a Connection to a Database

A connection to a database can be created using four different methodologies:

1. Using the Connection Wizard
2. Using the Server Explorer
3. Using the **Connection** class
4. Using specific database **Connection** objects

Enumerating through Instances of Microsoft SQL Server

One of the most useful data connection features of the .NET 2.0 Framework is the ability to easily set up SQL instance discovery for ease of application connectivity. The process of SQL instance discovery is as follows:

1. Instantiate the **Instance** property of the **SqlDataSourceEnumerator** class (in the **System.Data.Sql** namespace).
2. Call the **GetDataSources** method.
3. Receive a returned DataTable with information on each SQL Server that is visible to the application on the network.

The DataTable information contains the following:

Column	Description
InstanceName	Server instance
IsClustered	Whether the server is part of a cluster
ServerName	Name of the server
Version	SQL Server version

Note: If a server is running a default instance, the InstanceName column will be blank.

Some SQL Servers may not show up in the DataTable for the following reasons:

1. Excessive network traffic
2. Security settings by administrators
3. Network timeouts
4. Firewall blockages
5. Protocol settings
6. Browser service not available

Opening an ADO.NET Connection to a Database

To open an ADO.NET connection, use the **Open** method. The process of opening a connection is as follows:

1. Ensure the **Connection** object contains a pointer string to a data source.
2. Ensure the **Connection** object contains appropriate connection information such as connection credentials.
3. State the changes in the connection that are set up to monitor the open connection.
4. Note that information is passed from the server such as warnings when the connection is open.

Connection events are used for steps 3 and 4 of the above:

StateChanged events are raised to indicate the current state, open or closed, of the database.

InfoMessage events provide information from the server concerning the open connection.

Closing an ADO.NET Connection to a Database by Using the Close Method of the Connection Object

The closing process of a data connection uses the **Connection** object's **Close** method. It is implemented in the following manner:

1. Create events for each closing object, such as a close button.
2. Implement the **Close** method to the event.

Note: When the **Close** method is called, all pending transactions and events are rolled back.

Protecting Access to Data Source Connection Details

Security is a critical concern to all database users and administrators. Although Windows authentication may be used, other security methods can help fortify security.

The easiest security method to use is to set the *Persist Security Information* keyword to *false*, which ensures that the credentials used in the connection are eliminated and not stored anywhere.

A second method is to encrypt the configuration file to prevent the interception of the data within that file.

Creating a Connection Designed for Reuse in a Connection Pool and Controlling a Connection Pool by Configuring Connection String Values Based on Database Type

Connection pools allow for the reuse of connections to reduce connection traffic, which fosters increased performance in the application. Connection pools are separated by the following:

1. Connection strings
2. Application domains
3. Processes

To set up connection pooling, the following must be set in the connection string:

Pooling = True

Each OLE DB provider has specific keywords to set up a connection pool, so a universal set of code is not possible.

The following are possible connection pooling connection string keywords with their default values:

1. **Connection Lifetime (0)**
2. **Connection Reset (True)**
3. **Enlist (True)**
4. **Load Balance Timeout (0)**
5. **Max Pool Size (100)**
6. **Min Pool Size (0)**
7. **Pooling (True)**

2.3 Creating, Adding, and Editing Data in a Connected Environment

Retrieving Data by Using a DataReader Object

The **DataReader** object facilitates the retrieval of data through the use of SQL statements. To retrieve data using the **DataReader** object, follow these steps:

1. Set the **CommandText** property of the **Command** object to appropriate SQL statement(s).
2. Place semicolons between the various SQL statements.
3. Call the **ExecuteReader** method of the **DataReader**.
4. Call the **NextResult** method of the **DataReader** to view each additional statement's returned results.

If there is another set of data, DataReader.NextResult = True; otherwise, it is False and no more data is to be retrieved.

Building SQL Commands

SQL commands can be built using the following methods:

1. In the Server Explorer
2. Directly in code

Creating Parameters for a Command Object

Parameters can be sent to:

1. SQL statements
2. Stored procedures

Parameters are used to change queries quickly in terms of specific criteria as they interact directly between the database and the application.

There are three types of parameters:

1. **Input**
2. **Output**
3. **InputOutput**

The **InputOutput** parameter both sends and receives data during the execution of a command.

Note: The **Input** parameter is the default.

To create a parameter, perform the following:

1. Declare an instance of the **Parameter** class.
2. Set the name of the instance to coincide with the parameter name.
3. Set the data type expected.
4. Set the parameter direction, as indicated in the previous section.
5. Add the parameter to the **Command** object.

Performing Database Operations by Using a Command Object

Command objects contain one or more parameters that facilitate the movement of data to and from the database and the application. **Command** objects allow SQL statements, functions, and stored procedures to be run against the database to perform certain specified actions.

Note: There is a **Command** object for each of the data providers, and the data provider must match the specific **Command** object for communications to occur between the data source and the application. Common **Command** objects are as follows:

1. **OdbcCommand**
2. **OleDbCommand**
3. **OracleCommand**
4. **SqlCommand**

Command objects can perform many functions, including:

1. Executing SQL statements
2. Executing stored procedures
3. Executing functions
4. Performing catalog operations
5. Returning scalar values
6. Returning XML data

Retrieving Data from a Database by Using a Command Object

Once the **Command** object is configured, it must be executed to perform its desired functionality. Four execution methods can be used:

Method	Description
ExecuteReader	Runs SQL statements that return rows
ExecuteScalar	Runs SQL statements that return a single value
ExecuteNonQuery	For performing update or catalog operations where no data is returned
ExecuteXmlReader	Runs SQL statements that return XML data

Performing Asynchronous Operations by Using a Command Object

An asynchronous process executes on a thread that is separate from the rest of the application. The **Command** object has several methods that are designed for asynchronous operations:

Method	Description
BeginExecuteNonQuery	Asynchronous **ExecuteNonQuery** method
BeginExecuteReader	Asynchronous **ExecuteReader** method
BeginExecuteXmlReader	Asynchronous **ExecuteXmlReader** method
EndExecuteNonQuery	Completes **BeginExecuteNonQuery**
EndExecuteReader	Completes **BeginExecuteReader**
EndExecuteXmlReader	Completes **BeginExecuteXmlReader**

2.4 Creating, Adding, and Editing Data in a Disconnected Environment

Creating a DataSet

A DataSet can be created using two methodologies:

1. Graphically (using the DataSet Designer or the Toolbox and form)
2. Programmatically in code, as follows:

Visual Basic:

```
Dim FinanceDataSet As New DataSet ("FinanceDataSet")
```

C#:

```
DataSet financeDataSet = new DataSet ("FinanceDataSet");
```

Note: There are two kinds of DataSet objects:

1. Typed
2. Untyped

Adding a DataTable to a DataSet

DataTables can be added to a DataSet by using the following code:

Visual Basic:

```
Dim NewAccounts as New DataTable
FinanceDataSet.Tables.Add(NewAccounts)
```

C#:

```
DataTable newAccounts = new DataTable();
financeDataSet.Tables.Add(newAccounts);
```

Adding a Relationship between Tables within a DataSet

To add a relationship between tables within a DataSet, do the following:

1. Declare the **DataRelation** objects.
2. Provide the affected columns to the **DataRelation** Constructor.
3. Add the relationship to the **Relations** collection of the DataSet.

Copying DataSet Contents

To create a copy of the DataSet contents construct the following code:

Visual Basic:

```
Dim CopyOf FinanceDataSet as New Dataset
CopyOfFinanceDataSet = FinanceDataSet.Copy
```

C#:

```
DataSet copyOfFinanceDataSet = new DataSet();
copyOfFinanceDataSet = financeDataSet.Copy;
```

Creating DataTables

Creating a DataTable is simple: It is created by declaring an instance of the **DataTable** object.

The following code creates a DataTable:

Visual Basic:

```
FinanceDataTable as New DataTable("FinanceData")
```

C#:

```
DataTable financeDataTable  =  new DataTable("FinanceData");
```

Creating and Using DataViews

DataViews, which are a part of **System.Data.DataView**, interact with **DataTable** objects to allow data to be viewed in data binding controls. DataViews allow for sorting, filtering, and modifying data.

To create a DataView, do the following:

1. Instantiate a new instance of DataView.
2. Pass the name of the table for the view.
3. Assign an instance of the DataView to **DataTable.DefaultView** for the default view.

Representing Data in a DataSet by Using XML

You can represent data in a DataSet using XML's **WriteXml** method. There are two basic methodologies:

1. Call **WriteXml** to save all of the tables and table contents as XML.
2. Call **WriteXml** to save data from a specific table.

There are two ways to transport the XML data:

1. Save to a file.
2. Write to a stream.

Generating DataAdapter Commands

DataAdapter commands can be generated in two ways:

1. Automatically by using the **CommandBuilder** object
2. Programmatically in code

Populating a DataSet by Using a DataAdapter

DataAdapters are objects that are specific to data providers and contain information that allows interaction with and between DataSets and **DataTable** objects.

A **SELECT** command is used by the DataAdapter to populate **DataTables** in a DataSet. The **SELECT** statement controls the **Insert**, **Delete**, and **Update** statements.

There are two basic ways to configure the DataAdapter commands:

1. Create **Command** objects and assign them to specific DataAdapter properties.
2. Use the **CommandBuilder** object.

Updating a Database by Using a DataAdapter

The **CommandBuilder** can generate **Insert**, **Delete**, and **Update** statements for the DataAdapter.

Note: The **SELECT** statement used with the **CommandBuilder** must return at least one of the following:

1. One primary key
2. One unique column

Resolving Conflicts between a DataSet and a Database by Using a DataAdapter

Two properties are used to assist in resolving conflicts between a DataSet and a database:

1. **MissingMappingAction**: Helps determine what to do when there is no matching table to fill in the DataSet.

   There are three enumerations for this property:

   1. **Error**
   2. **Ignore**
   3. **Passthrough**

2. **MissingSchemaAction**: Helps determine what to do when the expected schema is not present. There are four enumerations for this property:

   1. **Add**
   2. **AddWithKey**
   3. **Error**
   4. **Ignore**

Performing Batch Operations by Using DataAdapters

Batch processing can occur with a **DataAdapter** object through the use of the **DataAdapter.Update-BatchSize** property.

Note: Setting **UpdateBatchSize** to 0 forces the batch size to default to the largest batch size the server can tolerate.

2.5 Managing XML with the XML Document Object Model (DOM)

The XML Document Object Model (DOM) presents a hierarchical representation of a flat XML file, which facilitates easier navigation and searches as well as easier modification of the XML file.

The **XmlDocument** class is the primary class for the DOM.

XMLDocument class: Represents an XML document.

Namespace: System.Xml

Assembly: System.Xml (in system.xml.dll)

The **XmlDocument** class facilitates the reading of XML files into memory and allows for file manipulation through a collection of **XmlNode** objects, which allow for searches, data retrieval, and data manipulation.

Reading XML Data into the DOM

An XML document is read using the following procedures:

1. Create a new instance of the **XmlDocument** class.
2. Load the XML file using the **Load** method.
3. Validate the XML against a schema using the **XmlValidatingReader**.

An XML file can be loaded through the following:

1. **Stream**
2. **String**
3. **TextWriter**
4. **XmlWriter**

Note: To preserve all white space in a document, ensure the following setting is set:

XmlDocument.PreserveWhiteSpace = True

This setting compensates for the fact that only significant white space is preserved with the **Load** method. The **LoadXml** method is used if no white space is to be preserved at all.

Modifying an XML Document by Adding and Removing Nodes

The **XmlDocument** class allows for the adding, copying, and removing of nodes.

The following are major methods for creating new nodes:

1. **CreateComment**
2. **CreateDocumentFragment**
3. **CreateDocumentType**
4. **CreateElement**
5. **CreateProcessingInstruction**
6. **CreateTextNode**
7. **CreateXmlDeclaration**
8. **CreateWhitespace**
9. **CreateSignificantWhitespace**

The following are methods for node insertion:

1. **InsertBefore**
2. **InsertAfter**
3. **AppendChild**
4. **PrependChild**

Modifying Nodes within an XML Document

The methods of the **XmlDocument** class and the **XmlNode** class are used to modify nodes within an XML document. Through these classes the following can be achieved:

1. The value of existing nodes can be changed.
2. A set of child nodes can be replaced.
3. Individual nodes can be replaced.
4. Characters can be replaced (individual or a range).
5. Characters can be removed (individual or a range).
6. An attribute can be set.
7. An attribute can be updated.

Writing Data in XML Format from the DOM

Using the **XmlDocument.WriteTo** method, the content of an **XmlDocument** can be sent to the following:

1. File
2. Stream
3. Console

Working with Nodes in the XML DOM

XML nodes are listed in an ordered fashion through the **XmlNodeList** class. This class has the following methods to assist in the manipulation of XML nodes:

1. **XmlNode.ChildNodes**
2. **XmlDocument.GetElementsByTagName**
3. **XmlElement.GetElementsByTagName**
4. **XmlNode.SelectNodes**

Handling DOM Events

The **XmlDocument** class contains events that can be raised when specific changes occur to the node structure of the XML document. These events are basic and straightforward:

1. **NodeInserting**
2. **NodeInserted**
3. **NodeRemoving**
4. **NodeRemoved**
5. **NodeChanging**
6. **NodeChanged**

Modifying the XML Declaration

The **XmlDocument.CreateXmlDeclaration** method can be used to create an **XmlDeclaration** node, which is the first node in an **XmlDocument** instance.

The **CreateXmlDeclaration** method has the following parameters:

1. **Version** (1.0)
2. **Encoding** (represents the current encoding)
3. **Standalone** (document is independent of external resources)

Note: **Version** is always set to 1.0 because no other versions are currently supported.

Caution: The **XmlDeclaration** node must be set in the first position in an **XmlDocument** class; otherwise, an exception will be thrown.

2.6 Reading, Writing, and Validating XML by Using the XmlReader Class and the XmlWriter Class

The **XmlReader** and **XmlWriter** classes are derived from the **System.Xml** namespace and facilitate the rapid reading and writing of XML. They are both abstract classes that provide for basic parsing functionality.

Reading XML Data by Using the XmlReader Class

The **XmlReader** class provides rapid, forward-only access to an XML file. This access is in a non-cached mode, and the entire document is read from beginning to end. The **XmlReader** class has several major methods:

Method	Description
Create	Returns a new instance of **XmlReader**
GetAttribute	Gets the value of the attribute
MoveToAttribute	Moves to the attribute specified
MoveToElement	Moves to the element with the current attribute
MoveToFirstAttribute	Moves to the first attribute
MoveToNextAttribute	Moves to the next attribute
Read	Reads the next node in the stream
ReadInnerXml	Returns all content in the current node excluding start and end nodes
ReadOuterXml	Returns all content in the current node
Skip	Skips the children of the current node

Note: The primary method for reading the content of an XML document is the **Read** method, which returns a Boolean indicating the success of the read.

Reading XML Element and Attribute Content

The attributes of a specific node can be read using the **XmlReader** class. The class actually reads the attributes backwards, providing the only instance where this class actually does read backwards.

Several methods allow for the navigation of attribute content:

1. **MoveToAttribute**: Moves the reader to a specified attribute in the node.
2. **MoveToFirstAttribute**: Moves the reader to the first attribute in the node.
3. **MoveToNextAttribute**: Moves the reader to the next attribute in the node.

Once one of these methods is called, the following are indicated:

1. Name of the attribute (through the **Name** property)
2. Value of the attribute (through the **Value** property)

The **MoveToContent** method allows nodes that do not contain content, including the following, to be skipped over:

1. White space
2. Comments
3. Processing instructions

Note: The **MoveToContent** method ensures that the current node is a content node, because no action will be taken if the node is not a content node.

Reading XML Data by Using the XmlTextReader Class

The **XmlTextReader** class reads text in a file or stream. Since it is an implementation of the **XmlReader** class, all methods that apply to the **XmlReader** class also apply to the **XmlTextReader** class.

Reading Node Trees by Using the XmlNodeReader Class

The **XmlNodeReader** class, like the **XmlTextReader** class, is derived from the **XmlReader** class. Its purpose is to read the content of an **XmlNode** object.

Each **XmlNode** object is part of the DOM subtree, and the **XmlNodeReader** is designed to navigate that tree rapidly and effectively.

Validating XML Data by Using the XmlValidatingReader Class

In many cases it is necessary to validate the data against a certain schema. The **XmlValidatingReader** class performs this task. It reads the document and validates each element against a specified schema to ensure compliance.

Note: The **XmlValidatingReader** class acts as a wrapper to the current **XmlReader** instance.

The specific schema in this class is found in the **ValidationType** property, which has five different values:

1. **Auto**: Determines the most appropriate validation type
2. **DTD**: Document Type Definition
3. **None**: No validation type
4. **Schema**: XSD extensible schema definition
5. **XDR**: XML data reduced schema

Validation errors are handled through the **ValidationEventArgs** class, which has the following properties:

1. **Exception**
2. **Message**
3. **Severity**

Writing XML Data by Using the XmlWriter Class

The **XmlWriter** class allows for the writing of XML to the following:

1. File
2. Console
3. Stream
4. Programmer-specified output

An instance of **XmlWriter** can create an instantiation of **XmlWriterSettings**, which allows for the setting of the format of XML that is being written by the **XmlWriter**.

The **XmlWriterSettings** class has the following important properties:

1. **Indent**
2. **IndentChars**
3. **NewLineChars**
4. **NewLineHandling**
5. **NewLineOnAttributes**

To write elements to file outputs, the **XmlWriter** class is used. The following are the primary element-writing methods of the **XmlWriter**:

1. **WriteComment**
2. **WriteElementString**
3. **WriteEndElement**
4. **WriteFullEndElement**
5. **WriteStartDocument**
6. **WriteStartElement**

Note: The **XmlWriter** class can write simple or complex elements as follows:

Simple Elements: Use the **WriteElementString** method.

Complex Elements: Use the **WriteStartElement** and **WriteEndElement** methods together.

To write to attributes, use the **WriteAttributeString**. However, note that this method can only write to elements that were created by the **WriteStartElement** method.

3.0 Printing in Windows Forms

3.1 Managing the Print Process by Using Print Dialogs

Printing in the .NET 2.0 Framework

The .NET 2.0 Framework simplifies the often complex task of implementing printing functionality in an application. By using several built-in classes and methods, the programmer can develop complex printing functions quickly and easily that allow users to have the printing control they need.

Configuring Windows Forms Print Options at Run Time

The primary class used when implementing printing in a Windows Form is the **PrinterSettings** class, which has several important components.

PrinterSettings Class

Namespace: System.Drawing.Printing

Assembly: System.Drawing (in System.Drawing.dll)

Most of the **PrinterSettings** class is implemented automatically through the use of various printing components:

Component	Description
PrintDocument	Defines an object that sends output to a printer. The object is reusable.
PageSetupDialog	Implements a dialog box that facilitates users' choices of page settings including margins.
PrintDialog	Instantiates a print dialog box that configures a **PrintTicket** and a **PrintQueue**.

The **PrintDialog** component has several important properties that give the user printing functionality within the Windows Forms:

Property	Description
AllowCurrentPage	Current Page button status
AllowPrintToFile	Print to File box status
AllowSelection	Selection option status
AllowSomePages	Pages option status
Document	Indicates which **PrintDocument** is associated with the current **PrintDialog** box
PrinterSettings	The modifiable user settings with the current selected printer
PrintToFile	Print to File checkbox status

The programmer can manually program the status of the selected **PrintDialog** properties within the application. For example, if the programmer does not want the user to be able to print to file, the following code can be implemented:

Visual Basic:

```
PrintDialogX.AllowPrintToFile = False
```

C#:

```
printDialogX.AllowPrintToFile = false;
```

Changing the Printers Attached to a User's Computer in Windows Forms

The **PrintDialog** component, as described above, is the central authority on the selection of printers. When the printer selection is made by the application user, the **PrinterSettings** property is updated. No other action is necessary.

Configuring the PrintPreviewDialog Control

The **PrintPreviewDialog** control facilitates the viewing of a preview screen that allows the document to be reviewed before printing. It operates in the following manner:

1. The **Print** method of the **PrintDocument** class is called.
2. The output is redirected to the screen instead of the printer.
3. The user can change settings on the preview screen.
4. The user can print from the preview screen or close the screen.
5. If print is chosen, the **Print** method of the **PrintDocument** component sends output to the specified printer.

Setting Page Details for Printing by Using the PageSetupDialog Component

The **PageSetupDialog** component facilitates the selection options of the pages for the current print job. This component has several important properties:

Property	Description
AllowMargins	Margins are enabled or disabled.
AllowOrientation	Landscape or portrait are enabled or disabled.
AllowPaper	Paper source and size selections are enabled or disabled.
AllowPrinter	Printer button is enabled or disabled.
Document	The PrintDocument associated with the current print job.
MinMargins	The minimum margins that the user is permitted to select.

Note: The **PageSetupDialog** must be associated with a specific instance of the **PageSettings** class in order to show the **PageSetupDialog** box.

3.2 Constructing Print Documents

Configuring the PrintDocument Component

The **PrintDocument** component is the representative of the current document of the current print job. It cannot actually be seen at run time, but it runs in the background to facilitate the printing of a document.

The **PrintDocument** component starts the **Print** method and at the same time raises one or more **PrintPage** events, which is the primary event in the printing process. The event handler contains several **PrintPageEventArgs**, as follows:

Properties	Description
Cancel	If set to **True**, the print job is cancelled.
Graphics	Renders the drawing surface content to the printed page.
HasMorePages	Indicates if more pages are to be printed in the document.
MarginBounds	Calls the **Rectangle** object to represent the page within the set margins.
PageBounds	Calls the **Rectangle** object to represent the total area of the current page.
PageSettings	Gets the **PageSetting** object for the current page to be printed.

Printing a Text Document in a Windows Form

You can print text by using the **Graphics** object of the **PrintPageEventArgs**. The **DrawString** method of the **Graphics** object renders a string of text to the printer.

The following must be specified when sending text to the printer through this method:

1. The specific font to be used
2. The text
3. The **Brush** object
4. The printing coordinates

Printing Graphics in a Windows Format

The **PrintPageEventArgs Graphics** object is used to render graphics to the printer in the same way that graphics are rendered to the screen.

To print simple graphics, the **Graphics** object is used.

To print complex graphics, the **GraphicsPath** object is used.

To print images, the **DrawImage** method of the **Graphics** object is used.

Alerting Users to the Completion of a Print Job

The **PrintDocument EndPrint** event is used to create a notification to the user that the current print job is completed. Once a print job is completed, the **PrintDocument.EndPrint** event is raised.

Enabling Security for Printing in Windows Forms

The .NET 2.0 Framework considers printing a secured activity; therefore, the **PrintingPermission** class is used to control printing permissions through the use of four **PrintingPermissionLevel** values:

Values	Description
AllPrinting	Access to the specified printer is not restricted.
DefaultPrinting	Enables printing to the default printer.
NoPrinting	Blocks access to the printer.
SafePrinting	Printing is only allowed through a printer dialog box.

3.3 Creating Customized PrintPreview Components

Setting the Document Property to Establish the Document to Be Previewed

In order to create customized print preview applications, the **PrintPreviewControl** is used in lieu of the **PrintPreviewDialog** that was demonstrated in the previous section. The **PrintPreviewControl** calls the **PrintDocument.Print** method and sends the output to the control instead of the printer.

It has several properties:

Property	Description
AutoZoom	Determines whether the **Zoom** property is automatically adjusted
Columns	Sets the pages that are displayed horizontally across the screen
Document	The current instance
Rows	Sets the pages to be displayed vertically across the screen
StartPage	Sets the first page to be displayed
UseAntiAlias	Sets the value that indicates whether anti-aliasing is used
Zoom	Sets the zoom level

The **Document** property is set to represent the **PrintDocument** that is currently being used and associates it with the **PrintPreviewControl**. The **PrintPreviewControl.Document** is set in the Properties window.

Setting the Columns and Rows Properties to Establish the Number of Pages That Will Be Displayed

Setting the **Columns** and **Rows** properties is a simple process. The following code demonstrates the ease of setting these two properties:

Visual Basic:

```
PrintPreviewControlX.Rows = 2
PrintPreviewControlX.Columns = 5
```

C#:

```
printPreviewControlX.Rows = 2;
printPreviewControlX.Columns = 5;
```

Setting the UseAntiAlias property

The **UseAntiAlias** property allows you to smooth the edges of drawings to improve their appearance. It is a Boolean value and can be set as follows:

Visual Basic:

```
PrintPreviewControlX.UseAntiAlias = True
```

C#:

```
printPreviewControlX.UseAntiAlias = true;
```

Setting the Zoom Property to Establish the Relative Zoom Level

The size of a page in the **Zoom** property of the **PrintPreview** control can be set based on 1.0 being the full size of the page. Values below 1.0 represent sizes lower than 100% and values above 1.0 indicate sizes above 100%. For example, a value of 0.5 indicates that the page will be represented at 50% of its size, whereas a value of 5 indicates that the page will be displayed at 500% of its original size.
The control can be set to 50% of the page size through the following code:

Visual Basic:

```
PrintPreviewControlX.Zoom = .5
```

C#:

```
printPreviewControlX.Zoom = .5;
```

The **AutoZoom** property allows the document to be automatically zoomed when the **PrintPreviewControl** is zoomed. Setting **AutoZoom** to **True** activates the auto-zoom capability.

Setting the StartPage Property

The **StartPage** property simply sets the first page to be displayed in the print preview. The following code demonstrates the **StartPage** property, setting the start page to 2:

Visual Basic:

```
PrintPreviewControlX.StartPage = 2
```

C#:

```
printPreviewControlX.StartPage = 2;
```

Note: This property can only be set at run time.

Adding Custom Methods and Events to a PrintPreviewControl

The **PrintPreviewControl** can be customized through the addition of methods and events programmed by the developer. This flexibility adds value to the application through the creativity of the developer. The following demonstrates how methods and events are added:

Visual Basic:

```
Public Class NewPrintCapabilities
        Inherits PrintPreviewControl
        ' Add new method or event here
End Class
```

C#:

```
public class NewPrintCapabilities : PrintPreviewControl {
        // Add new method or event here
}
```

As can be seen, the creative possibilities are vast through the sub-classing of **PrintPreviewControl**.

4.0 Enhancing Usability

4.1 Performing Drag and Drop Operations

Application Usability

The human-computer interaction of an application is a critical dynamic that can ultimately determine the viability of the application in the eyes of the user. Therefore, the application's usability must be a major consideration when designing and coding.

The .NET 2.0 Framework provides many features that assist in the development of a usable application. In turn, the application will foster increased productivity as well as lower learning curves for users, resulting in a high demand for the application. The .NET 2.0 Framework makes it easy to implement and enhance usability while the original code is being developed, saving both time and programming effort.

Drag and Drop Operations

The concept of drag and drop stems from the origins of Windows. It is a feature that is expected from all Windows applications. Programmatically, drag and drop operations are event-driven operations.

The initiation of drag and drop events starts with the **DoDragDrop** method. This method determines the control under the cursor's current location on the screen. It also validates the drop target as a valid target. The **DoDragDrop** method returns a value from the **DragDropEffects** enumeration, which is the final result of the drag and drop operation. The returned value will be one of the six **DragDropEffects** values found on the next page:

DragDropEffects	Description	Value
All	Data is copied, removed from the drag source, and then scrolled in the drop target.	–2147483645
Copy	Data is copied to the drop target.	1
Link	Data from the source is linked to the drop target.	4
Move	Data is moved to the drop target.	2
None	The data is not accepted by the drop target.	0
Scroll	Scrolling will or already is happening in the drop target.	–2147483648

The changes in the position of the mouse are tracked by the **DoDragDrop** method:

Raised Event	Action
DragDrop	The mouse button is released over the target control.
DragEnter	The mouse enters a new target control area.
DragLeave	The mouse moves out of the current window.
DragOver	The mouse moves but it stays within the current target control.
GiveFeedback	The drop target is valid.
QueryContinueDrag	Determines whether to continue, drop, or cancel the drag.

The **QueryContinueDrag** event can raise three different possibilities:

1. **DragAction.Continue**
2. **DragAction.Drop**
3. **DragAction.Cancel**

Note: The **DragOver** and **GiveFeedback** events should be used together so that up-to-date position information is given as the curser moves over the target.

Performing Drag and Drop Operations between Applications

No special code is required to facilitate drag and drop operations between applications, thanks to the inherent functionality of the .NET 2.0 Framework. The only action taken upon the target control is to sync it to the corresponding **DoDragDrop** method in the following two ways:

1. A drag effect listed in the **DoDragDrop** method call must be present.
2. The data format must be the same as in the **DoDragDrop** method call.

Performing Drag and Drop Operations by Using a TreeView Control

The **TreeView** class allows for the displaying of hierarchical data in the form of a tree.

TreeView Class

Namespace: System.Web.UI.WebControls

Assembly: System.Web (in system.web.dll)

TreeView controls are composed of nodes, each of which is represented by a **TreeNode** object.

The implementation of drag and drop operations is different than in regular operations. Drag operations are implemented on individual **TreeNodes**. Once implemented, the **ItemDrag** event is raised, passing an instance of **ItemDragEventArgs** (which references the **TreeNode**) to the handling method. The reference to the **TreeNode** is then copied directly to the **DataObject** in the **DoDragDrop** method.

4.2 Implementing Globalization and Localization for a Windows Forms Application

The globalization and localization features of the .NET 2.0 Framework allow for the displaying of data that will be acceptable to various cultures, making the applications, in essence, globally ready.

Globalization: Formatting data with formats that are culturally appropriate and acceptable.

Localization: Retrieving data that is culturally appropriate and acceptable.

The culture of the application can be changed through the use of the **CultureInfo** class, which provides information about a specific culture.

CultureInfo Class

Namespace: System.Globalization

Assembly: mscorlib (in mscorlib.dll)

The information specified by the **CultureInfo** class includes the following:

1. Language
2. Sublanguage
3. Country/Region
4. Calendar
5. Conventions

The **CultureInfo** class designates a unique name for each culture based on the RFC 1766 standard for Windows 2000 and XP, and RFC 3066 for Windows Vista.

The current culture can be set in the application by setting the **CurrentThread.CurrentCulture** property, as demonstrated in the following code, which sets the current culture to traditional Chinese:

Visual Basic:

```
System.Threading.Thread.CurrentThread.CurrentCulture = New _
    System.Globalization.CultureInfo("zh-Hant")
```

C#:

```
System.Threading.Thread.CurrentThread.CurrentCulture =
    new System.Globalization.CultureInfo("zh-Hant");
```

Several languages, such as Arabic, implement writing from right to left instead of left to right. The .NET 2.0 Framework allows for an easy transition in text direction through the **RightToLeft** property, which can be set to the following:

1. **Yes**
2. **No**
3. **Inherit**

The use of the **Inherit** setting allows the **RightToLeft** property to determine the value set by the parent control. This value will determine whether the **Yes** or **No** setting will apply.

4.3 Implementing Accessibility Features

Allowing users with a variety of physical needs to access and use the application is the heart of the accessible design paradigm. An accessible design begins in the initial development phase of the application and continues throughout the application coding process. At a minimum, the design should include the ability to use accessibility aids and to accept input and send output through a variety of methodologies in addition to the standard mouse and keyboard, allowing for user flexibility and ease of use.

The use of standard system settings is a major part of designing an accessible application. These settings include such aspects as font size, display sizes, system colors, icon size, and so forth. The **System.Drawing** namespace contains many of the classes that contain various user interface options.

The .NET 2.0 Framework provides a series of accessibility properties for Windows Forms Controls, which interact with many accessibility methodologies:

Property	Description
AccessibleDefaultAction-Description	Presents the default action description of a particular Windows control
AccessibleDescription	Presents a description that is sent to the accessibility aid
AccessibleName	Presents a name that is sent to the accessibility aid
AccessibilityObject	Presents information about a Windows control to the accessibility aid
AccessibleRole	Presents the role that is sent to the accessibility aid

Note: Accessibility properties can be set in the Properties window at design time.

4.4 Creating and Configuring Multiple Document Interface Forms

Creating MDI Parent and Child Forms

The .NET 2.0 Framework allows for the easy creation of applications that facilitate managing and working on several documents simultaneously. This is performed through the parent/child form model, where there is a single parent form that contains and manages several child forms. Multiple parent forms are permitted as well.

The primary, or main, form of the application is always the parent form, which is created before the child forms. This form is created by setting the **IsMDIContainer** property to **True** in the Properties window.

The child forms usually contain individual documents and data associated with those documents. Child forms are created after the parent form by adding subsequent forms to the application with their corresponding controls. The child forms are controlled by the parent form by creating a new instance of the child form in the parent form and then simply setting the **MdiParent** property to the parent form object.

Identifying the Active MDI Child Form

Identifying the active MDI child form is quite simple. The **ActiveMDIChild** property of the parent form can be set to reference the last form opened as follows:

Visual Basic:

```
Dim demoForm as Form
demoForm = Me.ActiveMDIChild
```

C#:

```
Form demoForm;
demoForm = this.ActiveMDIChild;
```

Arranging MDI Child Forms

In order to arrange the forms in an application, the **LayoutMdi** method is called, which takes parameters from the **MdiLayout** enumeration. These parameters align the child forms with the parent form in terms of cascading, tiling, or arranging icons, as follows:

Member	Description
ArrangeIcons	Icons are arranged.
Cascade	Windows are cascaded.
TileHorizontal	Windows are tiled horizontally.
TileVertical	Windows are tiled vertically.

Creating a Window List Menu for a MDI Application

A window list is a list of all of the current windows in the application, which allows users to select a window and have it activated in the parent form.

The implementation of a window list is simple. Just drag the **MenuStrip** component onto the parent form and create a top-level menu item. Then, set the **MdiWindowListItem** property in the Properties window.

Once configured, the child forms will populate the list entries.

4.5 Creating, Configuring, and Customizing User Assistance Controls and Components

The .NET 2.0 Framework provides many built-in components and controls that allow for the design of user-friendly applications that will enhance the usability of the application.

The following table contains the main user assistance controls:

Control	Description
ProgressBar	Allows for the visual indication of the progress of an operation through the **Maximum**, **Minimum**, **Step**, and **Value** properties
PropertyGrid	Allows the user to set the properties of controls at run time through the **PropertySort** enumeration values of **Alphabetical**, **Categorized**, **CategorizedAlphabetical**, and **NoSort**
StatusStrip	Allows for the display of application status information through the use of **ToolStrip** controls and **ToolStripItems**

The following table contains the main user assistance components:

Component	Description
ErrorProvider	Allows feedback for errors when an error condition appears for a specific control
ToolTip	Allows for the setup of tool tips for various controls that appear as pop-ups when the mouse hovers over a specific control
HelpProvider	Allows for the setting of Help properties with the **Help** namespace using the **HelpProvider** component

4.6 Persisting Windows Forms between Sessions

Property values are permitted to be persisted between user sessions in the .NET 2.0 Framework, and can be accessed and changed at run time. This feature helps eliminate redundant code by maintaining common settings such as font sizes, colors, and the like.

To persist a setting, the setting must be given a unique name and value in the application. It is then bound to a particular property that needs to persist between user settings.

The Settings Editor is used to persist property values through four properties:

1. **Name**
2. **Type**
3. **Scope**
4. **Value**

These settings can be accessed or changed at run time.

5.0 Implementing Asynchronous Programming Techniques to Improve the User Experience

5.1 Managing a Background Process by Using the BackgroundWorker Component

Asynchronous Programming

The inherent complexity of today's programs requires applications to run operations concurrently, which can cause the applications to slow, lock up, or become unresponsive. In many cases, time-intensive operations must compete for resources with the user interface, causing noticeable performance issues in the application. Such operations include complex scientific and financial calculations, complex query processing, and the downloading of large files.

The purpose of asynchronous programming techniques is to reduce the burden on the system by enabling the time-intensive operation to run asynchronously (in the background), which frees up resources to allow the application to operate in a normal, responsive manner. Visual Studio 2005 introduces a new component, the **BackgroundWorker** component, which significantly eases the burden of implementing asynchronous techniques within an application.

The BackgroundWorker Component

The primary facilitator of asynchronous programming in Visual Studio 2005 is the **BackgroundWorker** component, which allows designated operations to operate on a thread that is independent of the threads used by other processes of the application.

BackgroundWorker Class

Namespace: System.ComponentModel

Assembly: System (system.dll)

Methods of the BackgroundWorker Class:

Method	Description
ReportProgress	Raises the **ProgressChanged** event.
RunWorkerAsync	The primary method of the **BackgroundWorker** component. It executes background operations and starts the **DoWork** event on a separate, dedicated thread.
CancelAsync	Initiates a request to cancel a background operation.

Properties of the BackgroundWorker Class:

Property	Description
CancellationPending	Indicates that the **CancelAsync** method was called to cancel a background operation
IsBusy	Indicates whether an asynchronous operation is running
WorkerReportsProgress	Indicates whether there are report progress updates
WorkerSupportsCancellation	Indicates whether an asynchronous operation can be cancelled

Events of the BackgroundWorker Class:

Event	Description
DoWork	Triggered by a call to the **RunWorkerAsync** method. It causes the operation to operate on a dedicated thread.
ProgressChanged	Triggered by the **ReportProgress** method to report the progress of an asynchronous operation.
RunWorkerCompleted	Indicates completion or cancellation of the background operation.

Note: **BackgroundWorker** components cannot perform multithreaded operations in more than one **AppDomain**.

Running a Background Process

Running a background process is not complicated. Place the **BackgroundWorker** component onto the Windows form by simply dragging it from the Toolbox, then create the **DoWork** event followed by the code of the operation that needs to run asynchronously.

Whenever the operation needs to be run in an asynchronous manner throughout the program, the **RunWorkerAsync** method is called. For example, for a specific instance of the **BackgroundWorker** called X, which runs a certain operation X, the operation can be run asynchronously by implementing the following method calls:

Visual Basic:

```
BackgroundWorkerX.RunWorkerAsync ( )
```

C#:

```
backgroundWorkerX.RunWorkerAsync( );
```

Note: Many instances of **BackgroundWorker** can be implemented and called in an application. A unique instance of the component can be created for each process type, with a separate name for each component implementing the asynchronous process (for example, BackgroundWorker1, BackgroundWorker2, and so forth).

Announcing the Completion of a Background Process

The **RunWorkerCompleted** event is triggered upon the following three instances of a process:

1. The process is cancelled.
2. The process is completed.
3. An exception has been raised during the process.

The use of the **RunWorkerCompleted** event enables the program to notify the application user of the completion/cancellation/exception of the background process.

Canceling a Background Process

The **BackgroundWorker** component has a property called **WorkerSupportsCancellation**, which the programmer can implement manually into the application to allow the **BackgroundWorker** to support the cancellation of the asynchronous operation.

When coding the cancellation of the asynchronous operation, the **CancelAsync** method of the **BackgroundWorker** component is called, which then sets the **CancellationPending** property to **True**. When the asynchronous operation is processing, the **CancellationPending** property is set to **False**.

Reporting the Progress of a Background Operation

The **ReportProgress** method of the **BackgroundWorker** component allows the application to be notified of the progress of the asynchronous operation. This is useful for asynchronous operations that may be running for quite some time.

The **ReportProgress** method triggers the **ProgressChanged** event, which allows the percentage of progress to be received and reported by the **ReportProgress** method through a parameter sent to the method via the triggered event.

Requesting the Status of a Background Process

The **IsBusy** property of the **BackgroundWorker** component allows you to determine the status of the background process. The property has two possible values, **True** and **False**, where **True** indicates that a background process is currently running. This property can be used to test for a process and perform an operation based on the Boolean status of the process. For example, if the background process is running, and a need to cancel arises, the following code can be implemented:

Visual Basic:

```
If  BackgroundWorkerX.IsBusy
        BackgroundWorkerX.CancelAsync( )
End If
```

C#:

```
if backgroundWorkerX.IsBusy { backgroundWorkerX.CancelAsync( );}
```

5.2 Implementing an Asynchronous Method

Creating an Asynchronous Method

One way to invoke an asynchronous process, other than using the **BackgroundWorker** component, is through the creation of an asynchronous method. This can be done primarily through the use of a **Delegate**, which is a class that acts like a function pointer with type-safe characteristics.

A **Delegate** has the ability to call any method synchronously or asynchronously, making it a good candidate for an alternative to the **BackgroundWorker** component.

The following methods can be used by a **Delegate** to create an asynchronous process:

Method	Description
BeginInvoke	Starts a corresponding asynchronous method on an independent thread. Returns instances of **IAsyncResult**, which monitors the asynchronous call.
EndInvoke	Ends the asynchronous method thread and gets the results of that method for the application. Receives the result of **IAsyncResult** to determine when the method is finished.

The **BeginInvoke** method requires several parameters in many cases:

1. The parameters of the **Delegate** represented method
2. The **AsyncCallback Delegate**, which references the method to be called
3. The user-defined object that references the asynchronous call

In some instances, depending on the particular method, **"Nothing"** in Visual Basic or ***"null"*** in C# are passed in lieu of parameters.

With an asynchronous method, **BeginInvoke** and **EndInvoke** methods can be used on the same thread, or can be called on different threads, with the **EndInvoke** method being called once the operation is complete.

Creating a New Process Thread

Threads enable the programmer to exercise more control over asynchronous processes than with either the **BackgroundWorker** component or **Delegates**.

The central component is the **Thread** object, which is an independent thread of execution that can run concurrently with many threads. However, the more threads that are running the more system performance will be degraded.

The **Thread** object can call two methods:

1. **Thread.Start**
2. **Thread.Abort**

Once a thread is aborted it cannot be restarted.

Two issues to watch out for when creating and running asynchronous process threads are:

1. Deadlocks
2. Race conditions

Thread synchronization can be used to protect two or more threads from trying to access the same resource at the same time.

6.0 Developing Windows Forms Controls

6.1 Creating Composite Windows Forms Controls

Windows Forms Controls

The inclusion of forms controls allows the user of the application to easily operate within the form, giving the application true Windows functionality and ease of use. The .NET platform allows programmers to rapidly create Windows forms controls through one of four different control creation methods:

1. The use of built-in .NET controls
2. The creation of composite controls by the programmer
3. The creation of custom controls by the programmer
4. The creation of extended controls by the programmer

As you can, the built-in controls of .NET can be used by the programmer, or the programmer can combine or extend built-in controls, or create new ones altogether. This type of flexibility is a powerful feature of .NET and allows a wide range of new or unique controls to be added to a form for precision in functionality.

Creating Composite Windows Forms Controls

Composite controls are made by combining existing Windows controls. They inherit directly from the **UserControl** class.

UserControl Class

Namespace: System.Windows.Forms

Assembly: System.Windows.Forms in System.Windows.Forms.dll

UserControl extends **ContainerControl**, and thus inherits the standard positioning in a user control as well as the mnemonic handling code that is in a user control.

The creation of composite controls is quite simple, because they inherit directly from the **UserControl** class:

Visual Basic:

```
Public Class controlOne
        Inherits UserControl
        'insert functionality code
End Class
```

C#:

```
Public class controlOne : UserControl {
        // insert functionality code
}
```

Creating Properties, Methods, and Events for Windows Forms Controls

Properties
To add a property to a control, do the following:

1. Create the property definition.
2. Code the functionality.
3. Set the property value in a private variable.

Methods
Methods are added to controls in exactly the same manner that a method would be added to any type of class or form: within the class declaration of the Code window through the method declaration and the method body.

Events
To add an event to a control, Visual Basic and C# use differing methodologies.

Visual Basic: Use the **Event** keyword followed by the name and signature of the specific event.

C#: Specify an explicit delegate that indicates the signature and the event keyword.

Exposing Properties of Constituent Controls

Constituent controls are subordinate to composite controls. They are created the same way their parent composite controls are created and contain the functionality contained within the parent composite control.

Constituent controls are not available to classes within other assemblies, even within the same set of code; therefore, they must be either recoded in classes within other assemblies or exposed in other assemblies. This can be done by wrapping the constituent control in a property declaration and then having the composite control's code implemented through the get and set values of the constituent control's property.

Creating Custom Dialog Boxes

The purpose of a dialog box is to prompt the user to input information into the application. There are two types of dialog boxes:

1. Built-in .NET dialog boxes
2. Custom developed dialog boxes

Custom dialog boxes are created by using the Dialog Box template, which can be accessed through **Project Menu > Add New Item > Dialog Box template**. This template allows you to create dialog boxes that collect specifically formatted information from the application user.

There are two distinct ways a dialog box can be displayed:

1. *Modally*: Pauses the execution of the program until the dialog box is closed
2. *Modelessly*: Permits the program execution to continue while open

Configuring a Control to Be Invisible at Run Time

Making controls invisible at run time prevents them from being accessed by the application user; however, they are still available for use by the application itself. Making them invisible can be advantageous when user intervention in the application is not needed or warranted.

Configuring the invisibility of the control is a simple process in .NET. The **Visible** property of the control is simply set to **false**, as in the following examples:

Visual Basic:

```
controlOne.Visible = False
```

C#:

```
controlOne.Visible = false;
```

Note: To be certain that the control is set to be invisible at the start of the application, the control's **Load** event handler must have the control's **Visible** property set to **false** within the event handler.

Configuring a Control to Have a Transparent Background

The creation of a control with a transparent background is quite simple in .NET. The **BackColor** property of the control is set to **Color.Transparent**, as in the following code:

Visual Basic:

```
controlOne.BackColor = Color.Transparent
```

C#:

```
controlOne.BackColor = Color.Transparent;
```

6.2 Creating a Custom Windows Forms Control by Inheriting from the Control Class

Custom controls are customizable by the programmer and give the greatest degree of flexibility in terms of functionality of the control. Custom controls must be developed exclusively through the original code of the programmer.

Custom controls inherit directly from the **Control** class, which gives the custom control its basic functionality in the context of the current application. This includes the properties such as **Visible** and **BackColor** that were discussed previously. The targeted functionality of the custom control, however, must be provided by the programmer.

The namespaces used for custom control development are the following:

1. **System.Drawing**
2. **System.Drawing.Drawing2D**
3. **System.Drawing.Imaging**
4. **System.Drawing.Printing**
5. **System.Drawing.Text**

The most widely used classes to create custom controls are the **Graphics** class and the **Drawing** class, which present the drawing surface of the control and allow for added functionality through their many methods.

6.3 Creating an Extended Control by Inheriting from an Existing Windows Forms Control

Extended controls are custom-developed controls that build upon, or extend, current .NET controls. The benefit of creating extended controls is that the original functionality of the control is maintained with the addition of new methods, properties, and in some cases, a new appearance.

The process of extension is basically the same as extending any object in the .NET framework. A class is created, which inherits the control that is to be extended.

The following code demonstrates the creation of a class that inherits the control that is to be extended:

Visual Basic:

```
Public Class NewCheckBox
        Inherits System.Windows.Forms.CheckBox
End Class
```

C#:

```
public class NewCheckBox : System.Windows.Forms.CheckBox { }
```

The above new classes will allow the CheckBoxes to have the same behavior and properties as the **CheckBox** class, with the addition of new properties, methods, and appearances as provided by the programmer.

7.0 Configuring and Deploying Applications

7.1 Configuring the Installation of a Windows Forms Application by Using ClickOnce Technology

Deploying Applications

The best application in the world will create frustration for the user if it does not deploy rapidly, accurately, and correctly. The ease of deployment is paramount for a successful application.

In the past, users have been forced to deal with many issues that make the deployment of applications difficult, such as installing applications without admin rights, configuring the application to be able to receive security patches, installing updates and upgrades, and configuring the application to run on a shared drive or application server. These issues are compounded by the difficulty of getting past applications to launch from a remote location, such as a file server.

The .NET 2.0 Framework introduces a new technology to assist developers in creating a deployment strategy that is not only easy for the users to launch, but also easy for developers to implement in their software. This new technology is called ClickOnce technology.

ClickOnce technology provides several key benefits for deployment:

1. Applications are configured for easy updates or self-updates.
2. Users without administrative rights can install the applications.
3. The new application will install in an isolated manner, eliminating the need to share common files with already installed applications, which in the past posed many installation problems.
4. The number of user prompts during installation is reduced.
5. The application can be deployed effectively from a variety of sources such as:
    a. Network share
    b. Web page
    c. Local media (CD/DVD-ROM, USB drive)
6. The application can be installed whether the user is online or offline.
7. The application will install within the security mode of the network, or if installed on the user's local machine, full trust will be granted.

All applications developed in the .NET Framework 2.0 Windows Forms format will be able to be configured with the ClickOnce technology.

Installing a Windows Forms Application on a Client Computer or from a Server Using ClickOnce Technology

To set up ClickOnce for your new application, go to the *Solution Explorer* of the application, then right click the Windows Application project. From here, click on *Properties*, then click *Publish* to get the Publish screen. From this screen many options are available to set up the configuration required for the following:

1. Publishing location (Web site, FTP server, or file path)
2. Installation URL
3. Install mode and settings
    a. Online Only or Online/Offline
    b. Application Files Configurations
    c. Prerequisites
    d. Updates
    e. Options
4. Publish version number, with the ability to automatically increment the revision number with each publication

Each of these options can be set within the Publish screen without the need for additional lines of code, making the configuration of a deployment strategy quick and accurate, as well as consistent for each Windows Forms application.

The installation of a ClickOnce application is very simple from the standpoint of a user, and is also simple to explain in code documentation.

The three modes of installation have slightly different installation methods on a client computer.

Network Installation

From a network share, such as from a file server or a peer computer:

1. Connect to the share or server.
2. Locate the ClickOnce Application folder.
3. Double click the Setup icon.
4. Follow the Installation wizard.

Web Site Installation

From a Web site, either intranet or Internet:

1. Open the Web site.
2. Navigate to the Publish.htm page.
3. Click on the Install icon.
4. Follow the Installation wizard.

Local Client Installation

From the local client's CD/DVD-ROM or USB drive:

1. Open the media source.
2. Double click the Setup icon.
3. Follow the Installation wizard.

Note: ClickOnce applications can be published to the following locations:

1. FTP address
2. HTTP address
3. Network share location
4. Local file system

Configuring the Required Permissions on an Application by Using a ClickOnce Deployment

The configuration of permissions is accomplished through the Security tab, which is directly above the Publish screen's tab, as located in the previous section.

The default settings are as follows:

Install Location	Security Zone
Web site installs	Internet Security Zone
Network file share installs	Intranet Security Zone

Applications using ClickOnce security settings can be configured in the Security screen as:

1. Full trust applications
2. Partial trust applications

The Security screen allows you to set a variety of permissions through a scroll-down system, where individual permissions can be set manually or calculated by the .NET Framework to match the permissions required by the application.

7.2 Creating a Windows Forms Setup Application

Creating a Windows Forms Application Setup Project

Although ClickOnce technology provides an easy system of deployment configuration, in some cases, especially with specialized projects, a developer might want to create a customized deployment strategy.

Through the use of a Setup Project, a customized deployment solution can be created to do the following:

1. Create target directories.
2. Modify the registry.
3. Move and copy target files.
4. Add customizable installation actions.

The Setup Project solution is carried out through an .msi file, which will launch the setup wizard to install the application.

The Setup Project solution has six built-in editors to assist in the development of the Setup Project:

1. Custom Actions Editor
2. File System Editor
3. File Types Editor
4. Launch Conditions Editor
5. Registry Editor
6. User Interface Editor

Setting Deployment Project Properties

The properties of Setup Projects are configured in the Properties window. The developer can set a more than 20 properties in this window.

There are, however, two properties that should never be altered by the developer in this window:

1. **ProductCode**
2. **UpgradeCode**

Configuring a Setup Project to Add Icons During Setup

The addition of an icon is a simple process. The File System Editor is used to link a chosen icon to the application during the installation.

1. From the File System Editor, select a folder and then select Add > File.
2. In the opened folder, choose the .ico file and click Add.
3. Now select the shortcut in the File System Editor.
4. Browse to the icon and select the icon design required for the application.

Configuring a Conditional Installation Based on Operating System Versions

The installation of an application can include a provision to check for the current version of the client's operating system. This check can be used to allow for or abort an installation based on the current operating system at hand.

The system property *Version NT* is used for checking the current operating system.

VersionNT is an integer that is calculated through the use of the formula:

MajorVersion * 100 + MinorVersion

Thus, all versions of Windows that are Windows 2000 and later would be listed as:

VersionNT>=500

This system property prevents the installation of applications on legacy operating systems, sparing the user the frustration of attempting to load an application that is not designed for that legacy operating system.

Configuring a Setup Project to Deploy the .NET Framework

.NET Framework 2.0 must be installed for applications created with Visual Studio 2005 to run.

.NET Framework 2.0 can be configured to be a part of the installation prerequisites. By default, it is configured to be installed in the Setup Project deployment configuration.

7.3 Adding Functionality to a Windows Forms Setup Application

Adding a Custom Action to a Setup Project

The use of custom actions is conducive to a highly customizable setup project installation. This leads to an advanced application installation.

The custom action code must be within an **Installer** class, which is the base class for all custom installers in .NET Framework 2.0.

Note: The **Installer** class must be inherited in the application code.

Installer Class

Namespace: System.Configuration.Install

Assembly: System.Configuration.Install (in system.configuration.install.dll)

Methods of the Installer Class:

Method	Description
Install	Performs the application installation
Commit	Completes the installation transaction
Rollback	Restores the pre-installation state of the system
Uninstall	Removes the installation

Note: The **RunInstallerAttribute** must be added to the new class and set to **true**.

Adding Error Handling Code to a Setup Project for Custom Actions

Errors must be handled in the custom Setup Project environment because the custom actions are executables. Therefore, error handling code must be incorporated into the custom setup to avoid having the application crash if an error is encountered.

The basic error handling techniques are as follows:

1. Use of **Try**/**Catch** blocks
2. Throwing of an **InstallException**

The **InstallException** will allow the current installation to be rolled back to its original state.