DOT NET PROGRAMMING

CHAPTER – 4 : Introducing windows forms

SESSION – 29 : Overview of the system. windows. Forms Namespaces,

Overview of the system. windows.

This chapter introduces us to the System.Windows.Forms namespace. Here, we learn how to build a highly stylized main window (e.g., a Form-derived class). In the process, we learn about a number of window-related types, including MenuItem, ToolBar, StatusBar, and Application. This chapter also introduces how to capture and respond to user input (i.e., handling mouse and keyboard events) within the context of a GUI environment.

A Tale/Story of Three GUI Namespaces :

- The .NET platform provides three GUI toolkits, known as "Windows Forms," "Web Forms," and "Mobile Forms."

- The System.Windows.Forms namespaces contain a number of types that allow us to build traditional desktop applications, as well as Windows-based applications that target handled computing devices (such as the Pocket PC).Windows Forms hides the raw windowing primitives from view, allowing us to focus on the functionality of our application using the familiar .NET type system.

- Web Forms is a GUI toolkit used during ASP.NET development. The bulk of the Web Form types are contained in the System.Web.UI.WebControls namespace. Using these types, we are able to build browser-independent front ends based on various industry standards (HTML, HTTP, and so on).

- The .NET version 1.1 ships with a new GUI-centric namespace, System.Web.UI.MobileControls, which allow us to build UI (user interface)s that target mobile devices (such as cellular phones). Not surprisingly, the programming model of Mobile control applications mimics the functionality found within ASP.NET.

- It is worth pointing out that while Windows Forms, Web Forms, and Mobile Forms technologies define a number of similarly named types (e.g., Button, CheckBox, etc.) with similar members (Text, BackColor, etc.), they do not share a common implementation and cannot be treated identically.

Overview of the System.Windows.Forms Namespace :

- The System.Windows.Forms is composed of a number of classes, structures, delegates, interfaces, and enumerations.
- The following table lists some of the core classes found within System.Windows.Forms.

Windows Forms Class	Description
Application	Using the members of Application, you are able to process windows messages, start and terminate a
ButtonBase, Button,	These classes (in addition to many others) represent
CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox	types that correspond to various GUI widgets.
Form	This type represents a main window, dialog box, or MDI child window of a Windows Forms application
ColorDialog, OpenFileDialog, SaveFileDialog, FontDialog, PrintPreviewDialog, FolderBrowserDialog	As you might expect, Windows Forms defines a number of canned dialog boxes. If these don't fit the bill, you are free to build your own custom dialogs
Menu, MainMenu, MenuItem, ContextMenu	These types are used to build top most and context sensitive (pop-up) menu systems.
Clipboard, Help, Timer, Screen, ToolTip, Cursors	Various utility types used to facilitate interactive GUIs
StatusBar, Splitter, ToolBar, ScrollBar	Various types used to adorn a Form with common child controls.

Interacting with the Windows Forms Types :
When we build a Windows Forms application, we may choose to write all the relevant code by hand (using Notepad) and send the resulting *.cs files into the C# compiler using the /target:winexe flag. Taking time to build some Windows Forms applications by hand is not only a great learning experience, but also helps us understand the code generated by various GUI wizards.

Another option is to build Windows Forms projects using the Visual Studio .NET IDE. To be sure, the IDE does supply a number of great wizards, starter templates, and configuration tools that make working with Windows Forms extremely simple. The problem with wizards, is that if we do not understand what the generated code is doing on our behalf, we cannot gain a true mastery of the underlying technology.

Prepping the Project Workspace : To begin understanding Windows Forms programming, let's build a simple main window by hand. Let us first create a new empty C# project workspace named "MyRawWindow" using the VS .NET IDE. Next, insert a new C# class definition (resist the temptation to insert a new Windows Form class!) from the "Project | Add New Item..." menu option. Name this class MainWindow.

When we build a main window by hand, we need to use the Form and Application types (at minimum), both of which are contained in the System.Windows.Forms.dll assembly. A Windows Forms application also needs to reference System.dll, given that some types in the Windows Forms assembly make use of types in the System.dll assembly. Add references to these assemblies now (shown in Figure).

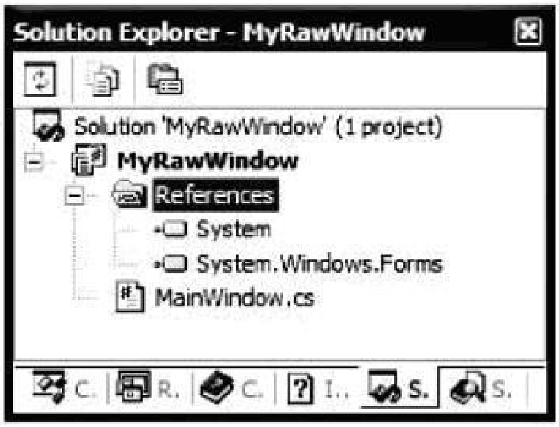


Figure : The minimal set of assembly references

Building a Main Window (By Hand) : In the world of Windows Forms, the Form class is used to represent any window in our application. This includes a topmost main window in an SDI (Single Document Interface) application, modeless and modal dialogs, as well as the parent and child windows of an MDI (Multiple Document Interface) application. When we are interested in creating a new main window, we have two mandatory steps:

- Derive a new custom class from System.Windows.Forms.Form.
- Configure the application's Main() method to call Application.Run(), passing an instance of our new Form-derived type as an argument.

With these steps in mind, we are able to update our initial class definition as follows:

```
using System;
using System.Windows.Forms;
namespace MyRawWindow
 {

        public class MainWindow : Form

        {

                public MainWindow(){ }

                // Run this application and identify the main window.
                public static int Main(string[] args)

                {

                        Application.Run(new MainWindow());
                        return 0;
                }

        }

 }
```

Here, we have defined with Main() method the scope of the class that represents the main window. If we prefer, one may wish to create a second class (we named ours, TheApp) which is responsible for the task of launching the main window:

```
namespace MyRawWindow
{

        public class MainWindow : Form

        {

                public MainWindow(){}

        }

        public class TheApp

        {

                public static int Main(string[] args)

                {

                        Application.Run(new  MainWindow()); retur
                        n 0;

                }

        }

}
```

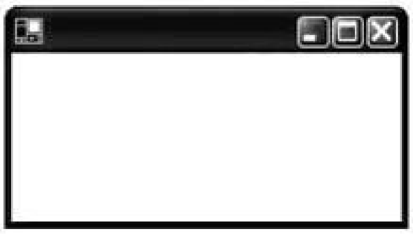
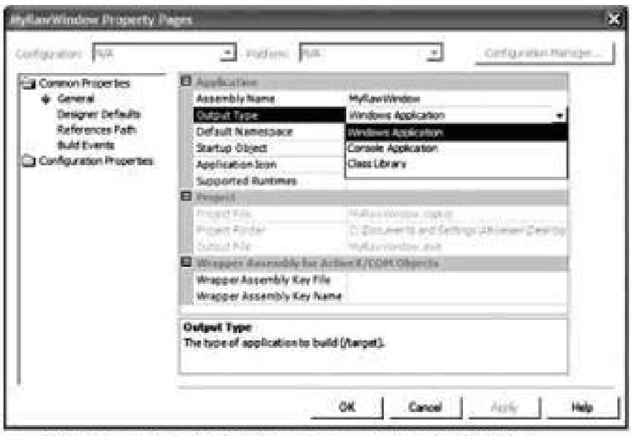


Figure : A simple main window a la Windows Forms

In either case, Figure shows a test run.

If we noticehow our MyRaw Window application has been launched, we should notice an irritating command window looming in the background. This is because one have not yet configured the build settings to generate a Windows *.exe application. To supply the /t:winexe flag from within the IDE, open the Project Properties



MyRawWindow Property Pages

Configuration: N/A Platform: N/A

Common Properties
- General
- Designer Defaults
- References Path
- Build Events

Configuration Properties

Application
- Assembly Name: MyRawWindow
- Output Type: Windows Application
  - Windows Application
  - Console Application
  - Class Library
- Default Namespace
- Startup Object
- Application Icon
- Supported Runtimes

Project

Wrapper Assembly for ActiveX/COM Objects
- Wrapper Assembly Key File
- Wrapper Assembly Key Name

Output Type
The type of application to build (/target).

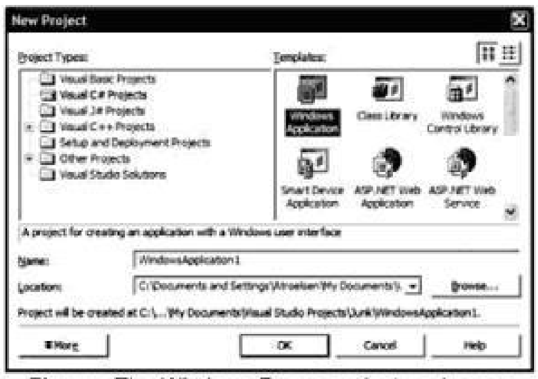
Figure : Specifying /t—winexe within VS .NET

window (just right-click the project icon from the Solution Explorer) and expand the "Common Properties | General" node. Finally, configure the "Output Type" property as "Windows Application" (as shown in Figure). When we recompile, the irritating command window will be gone and our application will run as a true-blue Windows app.

So, at this point we have a minimizable, maximizable, resizable, and closable main window (with a default system-supplied icon to boot!). To be sure, it is a great boon to the Win32 programmers of the world to forego the need to manually configure a WndProc function, establish a WinMain() entry point, and play the bits of a WNDCLASSEX structure.

Building a VS .NET Windows Forms Project Workspace : The benefit of building Windows Forms applications using Visual Studio

.NET is that the integrated tools can take care of a number of ordinary coding details by delegating them to a number of wizards, configuration windows, and so on. To illustrate how to make use of such assistance, close our current workspace. Now, select a new C# Windows Application project type (shown in Figure).



A project for creating an application with a Windows user interface

Name: WindowsApplication1

Location: C:\Documents and Settings\Atroelsen\My Documents\

Project will be created at C:\...\My Documents\Visual Studio Projects\Junk\WindowsApplication1.

Figure: The Windows Forms project workspace

When we click OK, we will find that we are automatically given a new class derived from System.Windows.Forms.Form [with a properly configured Main() method] and have references set to each required assembly (as well as some additional assemblies that one may or may not make use of). We will also see that we are given a design-time template that can be used to assemble the user interface of our Form (shown in Figure). Understand that as we update this design-time template, we are indirectly adding code to the associated Form-derived class (named Form1 by default).

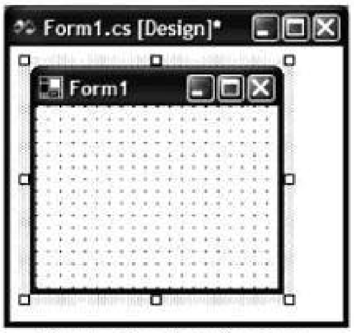


Figure : The Form Designer

Using the Solution Explorer window, we are able to alternate between this design-time template and the underlying C# code. To view the code that represents our current design, simply right-click the *.cs file and select "View Code". One can also open the code window by double-clicking anywhere on the design time Form; however, this has the (possibly undesirable) effect of writing an event handler for the Form's Load event.

Once we open the code window, we will see a class that looks very much like the following (XML-based code comments removed for clarity):

```
namespace VSWinApp
{
        public class Form1 : System.Windows.Forms.Form {
                private System.ComponentModel.Container components
                = null; public Form1()
                { InitializeComponent(); }
                protected override void Dispose( bool disposing )
                {
                        if( disposing )
                        {
                                if (components != null)
                                {
                                        components.Dispose();
                                }
                        }
                        base.Dispose(disposing);
                }
                #region Windows Form Designer generated
                code private void InitializeComponent() {

                        this.components = new System.ComponentModel.Contai
                        ner(); this.Size = new System.Drawing.Size(300,300);
                        this.Text = "Form1";
                }
                #endregion [ST
                AThread]
                static void Main()
                {
                        Application.Run(new Form1());
                }
        }
}
```

As we can see, this class listing is essentially the same code as the previous raw Windows Forms example. Our type still derives from System.Windows.Forms.Form, and the Main() method still calls Application.Run().

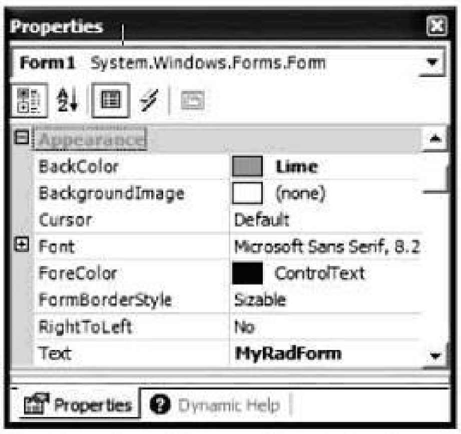
The major change is a new method nan

by the #region and #endregion preprocessor di

The Role of InitializeComponent() and Disp

is updated automatically by the Form

Designer to reflect the modifications



Properties

Form1 System.Windows.Forms.Form

⊟ Appearance	
BackColor	Lime
BackgroundImage	(none)
Cursor	Default
⊞ Font	Microsoft Sans Serif, 8.2
ForeColor	■ ControlText
FormBorderStyle	Sizable
RightToLeft	No
Text	MyRadForm

Figure : The Properties window provides design time editing support.

one make to the Form and its controls using the Visual Studio .NET IDE. For example, if we were to use the Properties window to modify the Form's Text and BackColor properties (shown in Figure), we would find that InitializeComponent() has been modified accordingly:

```
#region Windows Form Designer generated
code private void InitializeComponent() {
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 1
        3); this.BackColor = System.Drawing.Color.Lime;
        this.ClientSize = new System.Drawing.Size(292, 273);
        this.Text = "My Rad Form";
}
#endregion
```

The Form-derived class calls InitializeComponent() within the scope of the default constructor:

```
public Form1()
{
// Required for Windows Form Designer support
        InitializeComponent();
}
```

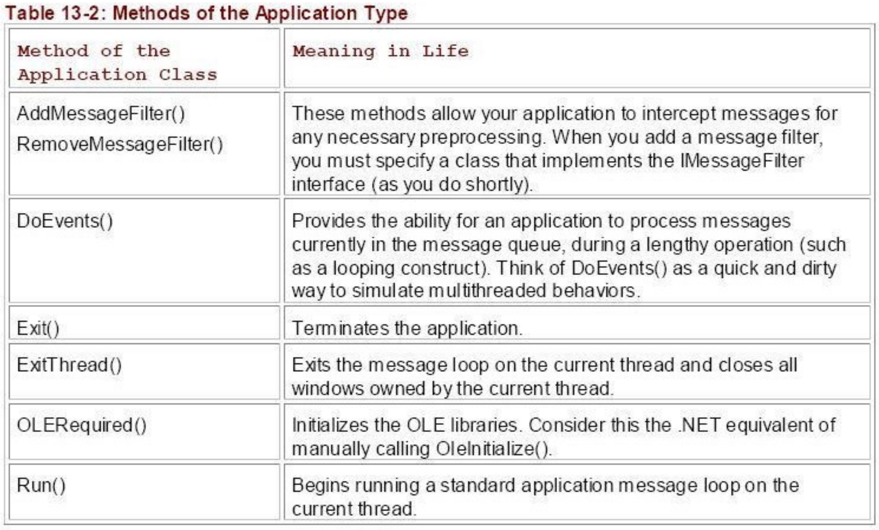
Do note that this helper method is simply a well-known member that VS .NET understands. If we moved all the code within this method into the Form's constructor, our application would run identically. Unlike other windowing toolkits, the wizard-generated code does not tie us to a particular IDE (VS .NET or otherwise). The final point of interest is the overridden Dispose() method. This method is called automatically when our Form is about to be destroyed, and is a safe place to free any allocated resources.

Application Class :

- The Application class defines members that allow us to control various low-level behaviors of a Windows Forms application.
- For example, the Application class defines a set of events that allow us to respond to events such as application shutdown and idle processing.

- The Application class also defines a number of static properties, many of which are read- only.

Table 13-2: Methods of the Application Type

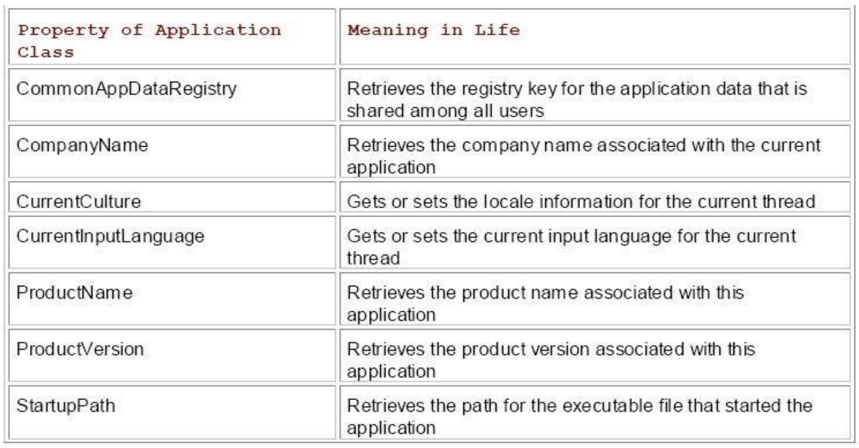


Method of the Application Class	Meaning in Life

AddMessageFilter()
RemoveMessageFilter()
	These methods allow your application to intercept messages for any necessary preprocessing. When you add a message filter, you must specify a class that implements the IMessageFilter interface (as you do shortly).

DoEvents()	Provides the ability for an application to process messages currently in the message queue, during a lengthy operation (such as a looping construct). Think of DoEvents() as a quick and dirty way to simulate multithreaded behaviors.
Exit()	Terminates the application.
ExitThread()	Exits the message loop on the current thread and closes all windows owned by the current thread.
OLERequired()	Initializes the OLE libraries. Consider this the .NET equivalent of manually calling OleInitialize().
Run()	Begins running a standard application message loop on the current thread.

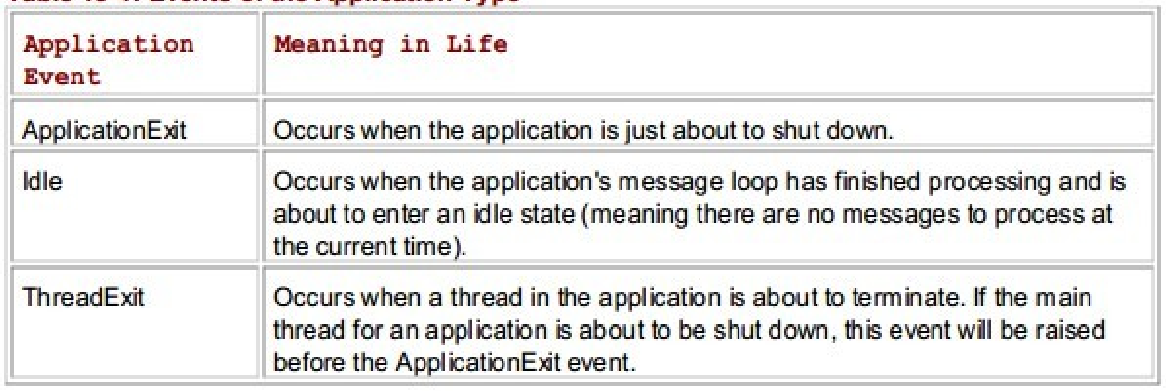
Table : Core Properties of the Application Ty



Property of Application Class	Meaning in Life
CommonAppDataRegistry	Retrieves the registry key for the application data that is shared among all users
CompanyName	Retrieves the company name associated with the current application
CurrentCulture	Gets or sets the locale information for the current thread
CurrentInputLanguage	Gets or sets the current input language for the current thread
ProductName	Retrieves the product name associated with this application
ProductVersion	Retrieves the product version associated with this application
StartupPath	Retrieves the path for the executable file that started the application

- Finally, the Application class defines the events shown in Table.

Table 13-4: Events of the Application Type



Application Event	Meaning in Life
ApplicationExit	Occurs when the application is just about to shut down.
Idle	Occurs when the application's message loop has finished processing and is about to enter an idle state (meaning there are no messages to process at the current time).
ThreadExit	Occurs when a thread in the application is about to terminate. If the main thread for an application is about to be shut down, this event will be raised before the ApplicationExit event.

The Anatomy of a Form :

- when we create a new window (or dialog box) we need to define a new class deriving from System.Windows.Forms.Form.
- This class gains a great deal of functionality from the types in its inheritance chain. F
- Form ultimately derives from System.Object. MarshalByRefObject defines the behavior to remote this type by reference, rather than by value.
- Thus, if we remotely interact with a Form across the wire, we are manipulating a reference to Form on remote machine.

- **The Component Class :** First base class of immediate interest is Component. Component type provides a canned implementation of the IComponent interface. This predefined interface defines a property named Site, which returns an ISite interface. Furthermore, IComponent inherits a single event from the IDisposable interface named Disposed

```
public interface IComponent : IDisposable
{
        // The Site property.
        public ISite Site { get;
        set; }
        // The Disposed event.
        public event EventHandler Disposed;
}
```

The ISite interface defines a number of methods that allow a Control to interact with the hosting container:

```
public interface ISite : IServiceProvider
{
        // Properties of the ISite interface.
        public IComponent Component
        { get; } public IContainer Container
        { get; } public bool DesignMode
        { get; } public string Name { get; set; }
}
```

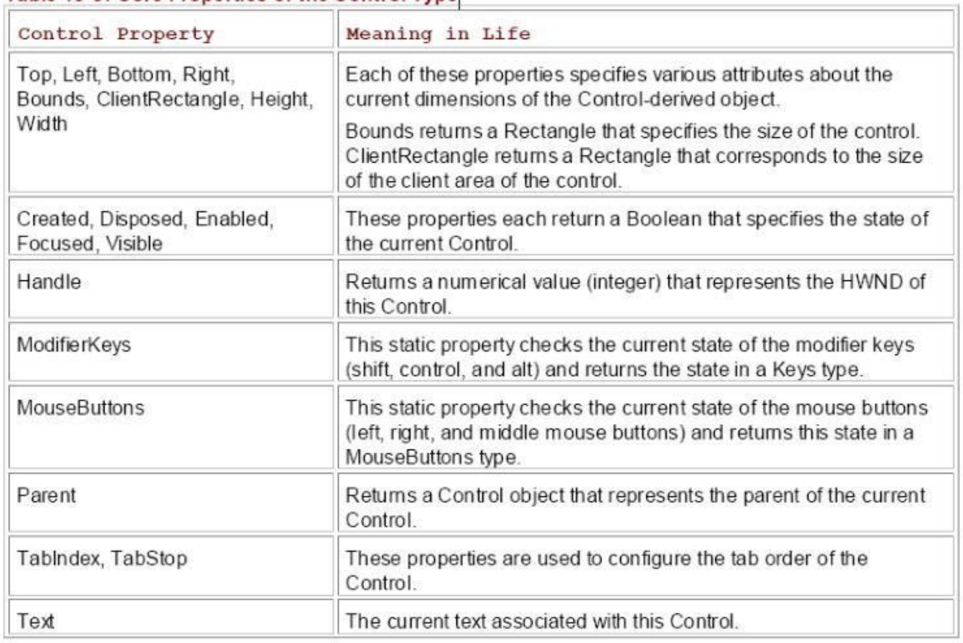
By and large, the properties defined by the ISite interface are only of interest to us if we are attempting to build a widget that can be manipulated at design time (such as a custom Windows Forms control).

Component also provides an implementation of the Dispose() method. When a Form has been closed, the Dispose() method is called automatically for the Form and for all widgets contained within that form.

- **The Control Class :** The next base class of interest is System.Windows.Forms.Control, which establishes the common behaviors required by any GUI-centric type. The core members of System.Windows.Forms.Control allow us to configure the size and position of a control, extract the underlying HWND (i.e., a numerical handle for a given window), as well as capture keyboard and mouse input.

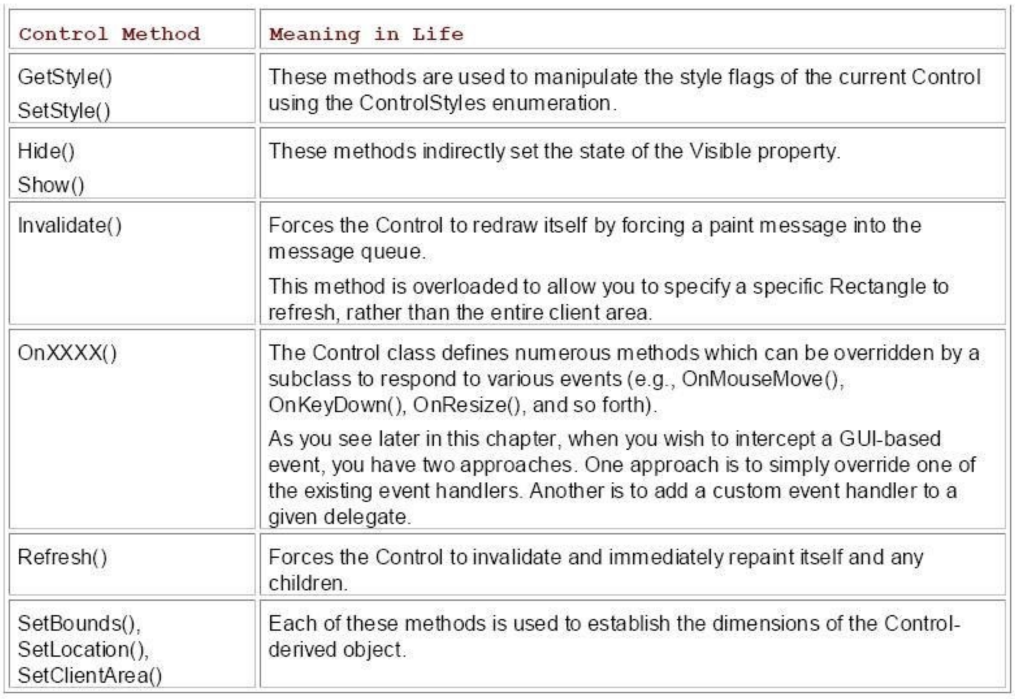
The Control base class also defines a number of methods that allow you to interact with any Control-derived type. A partial list of some of the more common members appears in Table.

Table 13-5: Core Properties of the Control Type



Control Property	Meaning in Life
Top, Left, Bottom, Right, Bounds, ClientRectangle, Height, Width	Each of these properties specifies various attributes about the current dimensions of the Control-derived object. Bounds returns a Rectangle that specifies the size of the control. ClientRectangle returns a Rectangle that corresponds to the size of the client area of the control.
Created, Disposed, Enabled, Focused, Visible	These properties each return a Boolean that specifies the state of the current Control.
Handle	Returns a numerical value (integer) that represents the HWND of this Control.
ModifierKeys	This static property checks the current state of the modifier keys (shift, control, and alt) and returns the state in a Keys type.
MouseButtons	This static property checks the current state of the mouse buttons (left, right, and middle mouse buttons) and returns this state in a MouseButtons type.
Parent	Returns a Control object that represents the parent of the current Control.
TabIndex, TabStop	These properties are used to configure the tab order of the Control.
Text	The current text associated with this Control.

Table : Select Members of the Control Type



Control Method	Meaning in Life

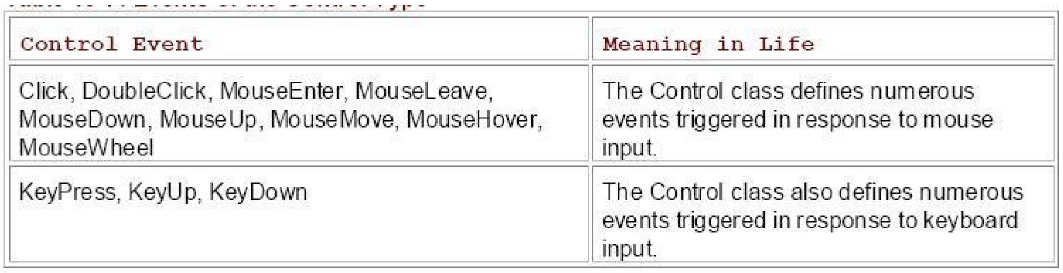
GetStyle()
SetStyle()
	These methods are used to manipulate the style flags of the current Control using the ControlStyles enumeration.

Hide()
Show()
	These methods indirectly set the state of the Visible property.

Invalidate()	Forces the Control to redraw itself by forcing a paint message into the message queue. This method is overloaded to allow you to specify a specific Rectangle to refresh, rather than the entire client area.
OnXXXX()	The Control class defines numerous methods which can be overridden by a subclass to respond to various events (e.g., OnMouseMove(), OnKeyDown(), OnResize(), and so forth). As you see later in this chapter, when you wish to intercept a GUI-based event, you have two approaches. One approach is to simply override one of the existing event handlers. Another is to add a custom event handler to a given delegate.
Refresh()	Forces the Control to invalidate and immediately repaint itself and any children.
SetBounds(), SetLocation(), SetClientArea()	Each of these methods is used to establish the dimensions of the Control-derived object.

Control Events : The Control class also defines a number of events that can logically be grouped into two major categories: Mouse events and keyboard events, shown in Table.

Table : Events of the Control Type



Control Event	Meaning in Life
Click, DoubleClick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseMove, MouseHover, MouseWheel	The Control class defines numerous events triggered in response to mouse input.
KeyPress, KeyUp, KeyDown	The Control class also defines numerous events triggered in response to keyboard input.

Function with the Control Class :

Control class does define additional properties, methods, and events beyond the subset we have examined.

However, to illustrate some of these core members, let's build a new Form type (also called MainForm) that provides the following functionality:

- Set the initial size of the Form to some arbitrary dimensions.
- Override the Dispose() method.
- Respond to the MouseMove and MouseUp events (using two approaches).
- Capture and process keyboard input.

```
// Need this for Rectangle definition.
using System.Drawing;
...
public class MainForm : Form
{
        public static int Main(string[] args)
        {
                Application.Run(new MainForm());
                 return 0;
        }
        public MainForm()
        {
                Top = 100;
                Left = 75;
                Height = 100;
                Width = 500;
                MessageBox.Show(Bounds.ToString(), "Current rect");
        }
}
```

When we run this application, we are able to confirm the coordinates of our Form via the Bounds property.

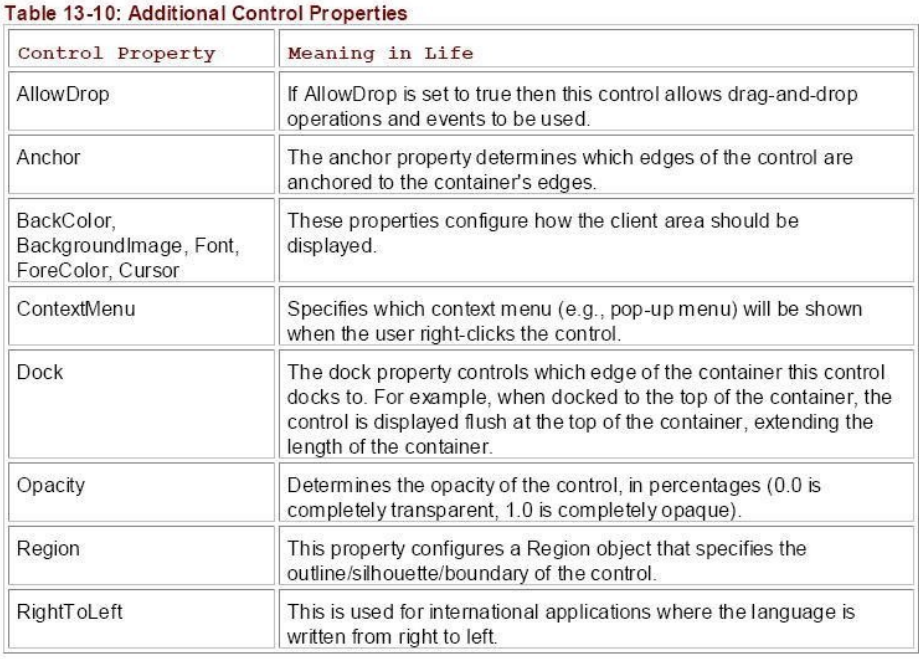
The Control Class Revisited :

The Control class defines further behaviors to configure background and foreground colors, background images, font characteristics, drag-and-drop functionality and support for pop-up context menus.

This class provides docking and anchoring behaviors for the derived types.

Perhaps *the most important duty of the Control class is to establish a mechanism to render images, text, and various geometric patterns onto the client area via a registered Paint event handler*.

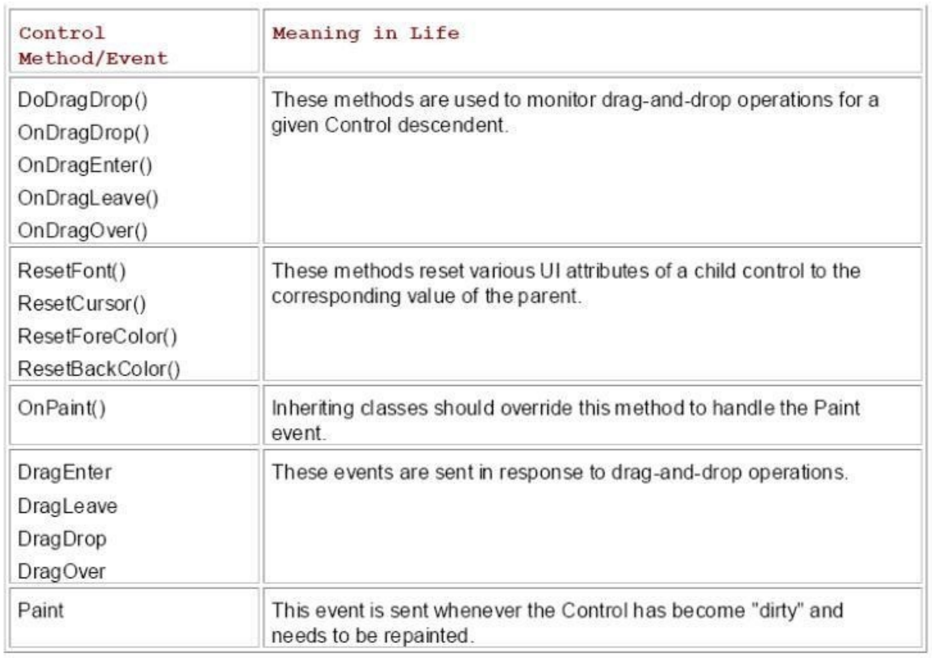
Table 13-10: Additional Control Properties



Control Property	Meaning in Life
AllowDrop	If AllowDrop is set to true then this control allows drag-and-drop operations and events to be used.
Anchor	The anchor property determines which edges of the control are anchored to the container's edges.
BackColor, BackgroundImage, Font, ForeColor, Cursor	These properties configure how the client area should be displayed.
ContextMenu	Specifies which context menu (e.g., pop-up menu) will be shown when the user right-clicks the control.
Dock	The dock property controls which edge of the container this control docks to. For example, when docked to the top of the container, the control is displayed flush at the top of the container, extending the length of the container.
Opacity	Determines the opacity of the control, in percentages (0.0 is completely transparent, 1.0 is completely opaque).
Region	This property configures a Region object that specifies the outline/silhouette/boundary of the control.
RightToLeft	This is used for international applications where the language is written from right to left.

The Control class also defines a number of additional methods and events used to configure how the Control should respond to drag-and-drop operations and respond to painting operations as shown in Table.

Table : Additional Control Methods



Control Method/Event	Meaning in Life

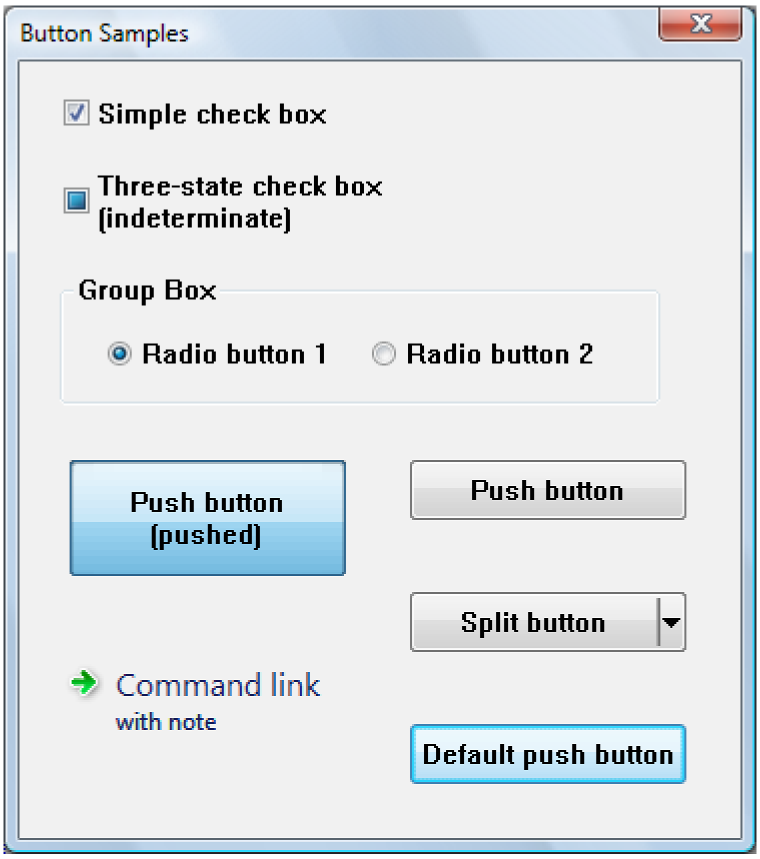
DoDragDrop()
OnDragDrop()
OnDragEnter()
OnDragLeave()
OnDragOver()
	These methods are used to monitor drag-and-drop operations for a given Control descendent.

ResetFont()
ResetCursor()
ResetForeColor()
ResetBackColor()
	These methods reset various UI attributes of a child control to the corresponding value of the parent.

OnPaint()	Inheriting classes should override this method to handle the Paint event.
DragEnter DragLeave DragDrop DragOver	These events are sent in response to drag-and-drop operations.
Paint	This event is sent whenever the Control has become "dirty" and needs to be repainted.

-

e :



The screen shot shows how buttons might appear in Windows Vista. The appearance will vary on different versions of the operating system, and according to the theme set by the user.

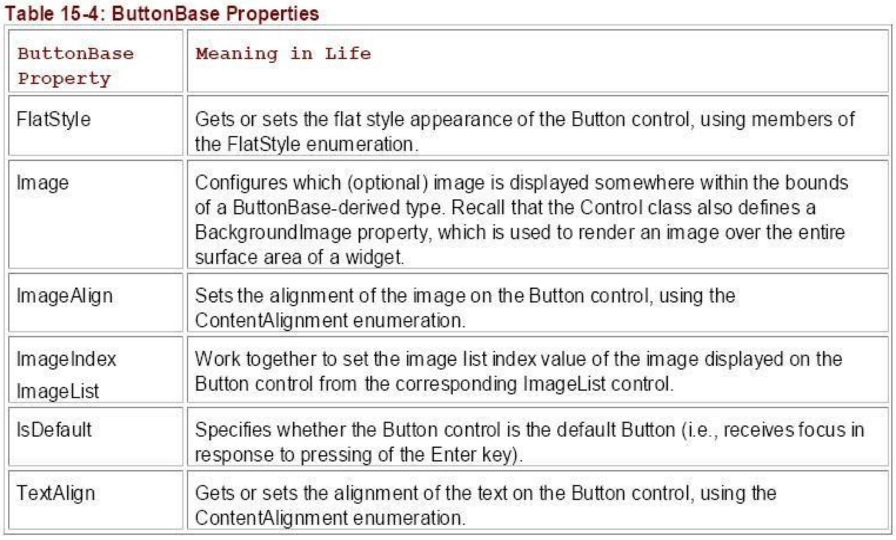
Note the following points about the illustration:

- The three-state check box is shown in the indeterminate state. When checked or unchecked, it looks like a normal check box.
- The large push button has been set to the pushed state programmatically (by sending the BM_SETSTATE message), so that it retains its appearance even when it is not being clicked.
- In the visual style shown, the background of the default push button (or another push button that has the input focus) cycles between blue and gray.

The role of the System.Windows.Forms.Button type is to provide a simple vehicle for user input, typically in response to a mouse click or key press.
The Button class immediately derives from an abstract type named ButtonBase, which provides a number of key behaviors for all Button-related types (CheckBox, RadioButton, and Button).

Below table describes some (but by no means all) of the core properties of ButtonBase.

Table 15-4: ButtonBase Properties



ButtonBase Property	Meaning in Life
FlatStyle	Gets or sets the flat style appearance of the Button control, using members of the FlatStyle enumeration.
Image	Configures which (optional) image is displayed somewhere within the bounds of a ButtonBase-derived type. Recall that the Control class also defines a BackgroundImage property, which is used to render an image over the entire surface area of a widget.
ImageAlign	Sets the alignment of the image on the Button control, using the ContentAlignment enumeration.
ImageIndex ImageList	Work together to set the image list index value of the image displayed on the Button control from the corresponding ImageList control.
IsDefault	Specifies whether the Button control is the default Button (i.e., receives focus in response to pressing of the Enter key).
TextAlign	Gets or sets the alignment of the text on the Button control, using the ContentAlignment enumeration.

The Button class itself defines almost no additional functionality beyond that inherited by the ButtonBase base class, with the key exception of the DialogResult property. A dialog box makes use of this property to return a value representing which Button was clicked (e.g., OK, Cancel, and so on) when the dialog box is terminated.

Fun with Buttons : To illustrate working with this most primitive of user input widgets, the following application uses the FlatStyle, ImageAlign, and TextAlign properties. The most interesting aspect of the underlying code is in the Click event handler for the btnStandard type (which would be the Button in the middle of the Form). The implementation of this method cycles through each member of the ContentAlignment enumeration and changes the Button's caption text and caption location based on the current value.

```
public class ButtonForm: System.Windows.Forms.Form {
// You have four Buttons on this Form. private
System.Windows.Forms.Button btnImage; private
System.Windows.Forms.Button btnStandard; private
System.Windows.Forms.Button btnPopup; private
System.Windows.Forms.Button btnFlat;
// Hold the current alignment value.
ContentAlignment currAlignment = ContentAlignment.MiddleCenter;
int currEnumPos = 0;
...
protected void btnStandard_Click (object sender, System.EventArgs e)
{
// Get all possible values of the ContentAlignment enum.
Array values = Enum.GetValues(currAlignment.GetType());
// Bump the current position in the enum.
// & check for wraparound.
currEnumPos++;
if(currEnumPos >=
values.Length) currEnumPos = 0;
// Change the current enum value. currAlignment =
(ContentAlignment)ContentAlignment.Parse(currAlignment.GetType()
, values.GetValue(currEnumPos).ToString());
// Paint enum name on Button.
btnStandard.Text =
currAlignment.ToString();
btnStandard.TextAlign = currAlignment;
// Now assign the location of the icon on btnImage...
btnImage.ImageAlign = currAlignment;
}
...
}
```

The output can be seen in Figure.

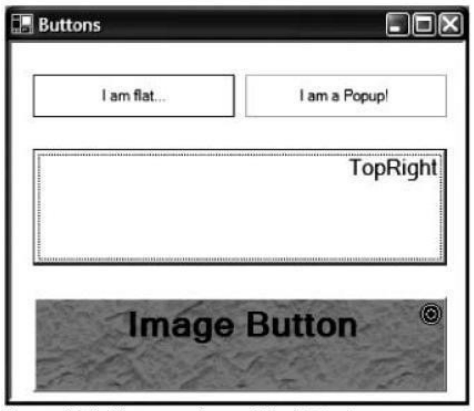


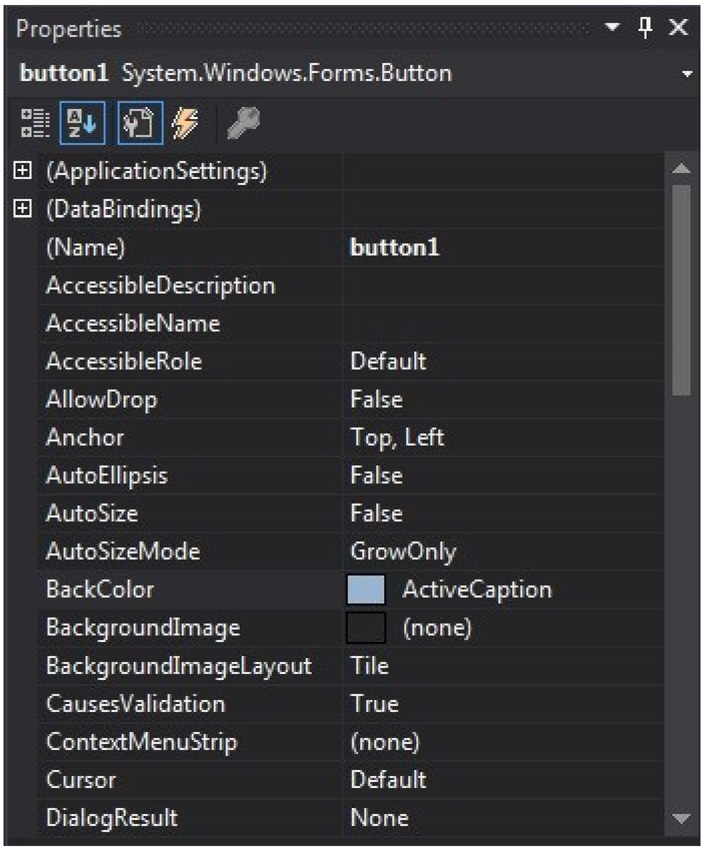
Figure 15-7: The many faces of the Button type

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         button1.Text = "Click Here";
      }

      private void button1_Click(object sender, EventArgs e)
      {
         MessageBox.Show("http://cshap.net-informations.com");
      }
   }
}
```

How to Call a Button's Click Event Programmatically

The Click event is raised when the Button control is clicked. This event is commonly used when no command name is associated with the Button control. Raising an event invokes the event handler through a delegate.

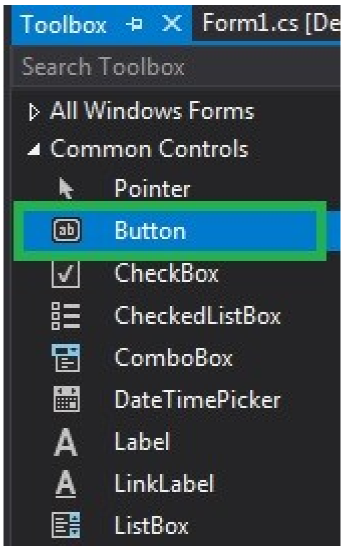
```
private void Form1_Load(object sender, EventArgs e)
{
   Button b = new Button();
   b.Click += new EventHandler(ShowMessage);
   Controls.Add(b);
}
private void ShowMessage(object sender, EventArgs e)
{
   MessageBox.Show("Button Click");
}
```



Properties

button1 System.Windows.Forms.Button

(ApplicationSettings)	
(DataBindings)	
(Name)	button1
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoEllipsis	False
AutoSize	False
AutoSizeMode	GrowOnly
BackColor	ActiveCaption
BackgroundImage	(none)
BackgroundImageLayout	Tile
CausesValidation	True
ContextMenuStrip	(none)
Cursor	Default
DialogResult	None



Property	Description
BackColor	Using BackColor property you can set the background color of the button.
BackgroundImage	Using BackgroundImage poperty you can set the background image on the button.
AutoEllipsis	Using AutoEllipsis property you can set a value which shows that whether the ellipsis character (…) appears at the right edge of the control which denotes that the button text extends beyond the specified length of the button.
AutoSize	Using AutoSize property you can set a value which shows whether the button resizes based on its contents.
Enabled	Using Enabled property you can set a value which shows whether the button

	can respond to user interaction.
Events	Using Events property you can get the list of the event handlers that are applied on the given button.
Font	Using Font property you can set the font of the button.
FontHeight	Using FontHeight property you can set the height of the font.
ForeColor	Using ForeColor property you can set the foreground color of the button.
Height	Using Height property you can set the height of the button.
Image	Using Image property you can set the image on the button.
Margin	Using Margin property you can set the margin between controls.
Name	Using Name property you can set the name of the button.
Padding	Using Padding property you can set the padding within the button.
Visible	Using Visible property you can set a value which shows whether the button and all its child buttons are displayed.

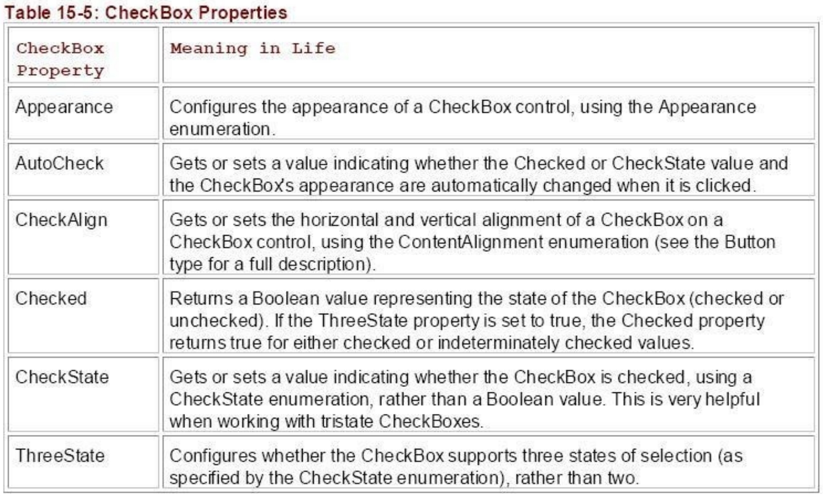
Important Events on Button

Event	Description
Click	This event occur when the button is clicked.
DoubleClick	This event occur when the user performs double click on the button.
Enter	This event occur when the control is entered.
KeyPress	This event occur when the character, or space, or backspace key is pressed while the control has focus.
Leave	This event occur when the input focus leaves the control.
MouseClick	This event occur when you click the mouse pointer on the button.
MouseDoubleClick	This event occur when you double click the mouse pointer on the button.
MouseHover	This event occur when the mouse pointer placed on the button.
MouseLeave	This event occur when the mouse pointer leaves the button.

***W*orking with CheckBoxes :**

The other two ButtonBase-derived types of interest are CheckBox (which can support up to three possible states) and RadioButton (which can be either selected or not selected). Like the Button, these types also receive most of their functionality from the Control base class. However, each class defines some additional functionality. First, consider the core properties of the CheckBox widget described in Table.

Table 15-5: CheckBox Properties



CheckBox Property	Meaning in Life
Appearance	Configures the appearance of a CheckBox control, using the Appearance enumeration.
AutoCheck	Gets or sets a value indicating whether the Checked or CheckState value and the CheckBox's appearance are automatically changed when it is clicked.
CheckAlign	Gets or sets the horizontal and vertical alignment of a CheckBox on a CheckBox control, using the ContentAlignment enumeration (see the Button type for a full description).
Checked	Returns a Boolean value representing the state of the CheckBox (checked or unchecked). If the ThreeState property is set to true, the Checked property returns true for either checked or indeterminately checked values.
CheckState	Gets or sets a value indicating whether the CheckBox is checked, using a CheckState enumeration, rather than a Boolean value. This is very helpful when working with tristate CheckBoxes.
ThreeState	Configures whether the CheckBox supports three states of selection (as specified by the CheckState enumeration), rather than two.

Checkboxes and Radio Buttons are way to offer your users choices. Checkboxes allow a user to select multiple options, whereas Radio Buttons allow only one.

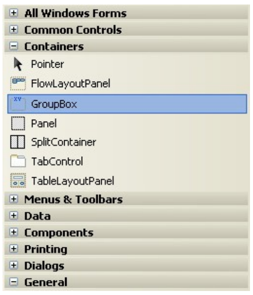
Let's see how to use them. Start a new project. When your new form appears, make it nice and big. Because Checkboxes and Radio Buttons are small and fiddly to move around, its best to place them on a Groupbox. You can then move the Groupbox, and the checkboxes and radio buttons will move with them.

Locate the Groupbox control in the Toolbox on the left, under Containers. It looks

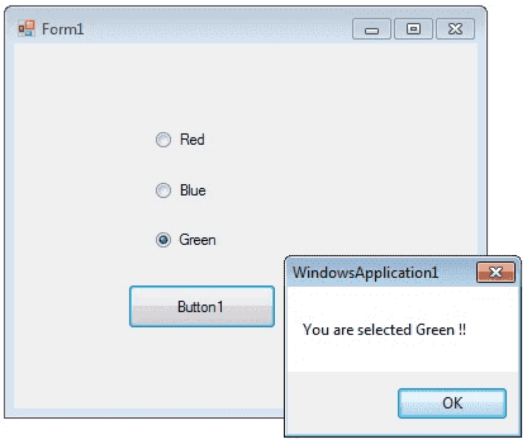
like this:

Draw one out on your form. Locate the Text property in the properties window on the right of C#.

Add a second Groupbox along side of the first one, and set the Text property as And Your Favourite Is?. Your form will then look like this:



Working with RadioButtons and GroupBoxes : The RadioButton type really requires little comment, given that it is (more or less) just a slightly redesigned CheckBox. In fact, the members of a RadioButton are almost identical to those of the CheckBox type.



You are selected Green !!

The only notable difference is the CheckedChanged event, which is fired when the Checked value changes. Also, the RadioButton type does not support the ThreeState property, as a RadioButton must be on or off.

Typically, multiple RadioButton objects are logically and physically grouped together to function as a whole. For example, if you have a set of four RadioButton types representing the color choice of a given automobile, you may wish to ensure that only one of the four types can be checked at a time. Rather than writing code programmatically to do so, use the GroupBox control. Like the RadioButton, there is little to say about the GroupBox control, given that it receives all of its functionality from the Control base class.

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         radioButton1.Checked = true;
      }

      private void button1_Click(object sender, EventArgs e)
      {
         if (radioButton1.Checked == true)
         {
            MessageBox.Show ("You are selected Red !! ");
            return;
         }
         else if (radioButton2.Checked == true)
         {
            MessageBox.Show("You are selected Blue !! ");
            return;
         }
         else
         {
            MessageBox.Show("You are selected Green !! ");
            return;
         }
      }
   }
}
```

Fun with RadioButtons (and CheckBoxes) : To illustrate working with the CheckBox, RadioButton, and GroupBox types, let's create a new Windows Forms Application named CarConfig, which will be extended throughout this chapter. The main Form allows users to enter in (and confirm) information about a new vehicle they intend to purchase. The order summary is displayed in a Label type once the Confirm Order button has been clicked. Below figure shows the user interface.

```
// Create your CheckBox.
checkFloorMats.Location = new System.Drawing.Point (16, 1
6); checkFloorMats.Text = "Extra Floor Mats";
checkFloorMats.Size = new System.Drawing.Size (136, 24); c
heckFloorMats.FlatStyle = FlatStyle.Popup;
```

```
// Add to Control collection.
this.Controls.Add (this.checkFloorMats);
```

Programmatically speaking, when we wish to place a widget under the ownership of a related GroupBox, we want to add each item to the GroupBox's Controls collection (in the same way you add widgets to the Form's Controls collection). To make things a bit more interesting, respond to the Enter and Leave events sent by GroupBox object as shown here:

```
// Yellow RadioButton.
radioYellow.Location = new System.Drawing.Point (96, 24);
radioYellow.Text = "Yellow";
radioYellow.Size = new System.Drawing.Size (64, 23);
// Green, Red, Pink RadioButtons configured in a similar
vein.
...
// Now build the group of radio items.
groupBox1.Location = new System.Drawing.Point
(16, 56); groupBox1.Text = "Exterior Color";
groupBox1.Size = new System.Drawing.Size (264, 88); group
Box1.Leave += new System.EventHandler (groupBox1_Leav
e); groupBox1.Enter += new System.EventHandler (groupBox
1_Enter); groupBox1.Controls.Add (this.radioPink); groupBox
1.Controls.Add (this.radioYellow); groupBox1.Controls.Add (th
is.radioRed); groupBox1.Controls.Add (this.radioGreen);
```

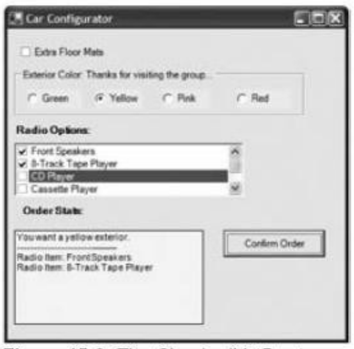
Understand, of course, that you do not need to capture the Enter or Leave events for a GroupBox. However, to illustrate, the event handlers update the caption text of the GroupBox as shown here:

```
// Figure out when the focus is in your group.
protected void groupBox1_Leave (object sender, System.EventArgs e)
{
groupBox1.Text = "Exterior Color: Thanks for visiting the group...";
}
protected void groupBox1_Enter (object sender, System.EventArgs e)
{
groupBox1.Text = "Exterior Color: You are in the group...";
```

The final GUI widgets on this Form (the Label and Button types) also need to be configured and inserted in the Form's Controls collection. The Label is used to display the order confirmation, which is formatted in the Click event handler of the order Button, as shown here:

```
protected void btnOrder_Click (object sender, System.EventArgs e)
{
// Build a string to display information. string
orderInfo = "";
if(checkFloorMats.Checked)
   orderInfo += "You want floor mats.\n";

if(radioRed.Checked)
   orderInfo += "You want a red exterior.\n";

if(radioYellow.Checked)
   orderInfo += "You want a yellow exterior.\n";

if(radioGreen.Checked)
   orderInfo += "You want a green exterior.\n";

if(radioPink.Checked)
   orderInfo += "Why do you want a PINK exterior?\n";
// Send this string to the Label.
infoLabel.Text = orderInfo;
}
```

Notice that both the CheckBox and RadioButton support the Checked property, which allows you to investigate the state of the widget. Recall that if you have configured a tristate CheckBox, you will need to check the state of the widget using the CheckState property.



You want a yellow exterior.
Radio Item: FrontSpeakers
Radio Item: 8-Track Tape Player

Figure 15-9: The CheckedListBox type

Examining the CheckedListBox Control : Now that you have explored the basic Button- centric widgets, let's move on to the set of list selectioncentric

types, specifically, CheckedListBox, ListBox, and ComboBox. The CheckedListBox widget allows you to group together related CheckBox options in a scrollable list control. Assume you have added such a control to your CarConfig application that allows the user to configure a number of options for regarding the automobile's sound system (as shown in Figure).

Like the controls examined thus far, the CheckedListBox type gains most of its functionality from the Control base class type. Also, the CheckedListBox type inherits additional functionality from its direct base class, ListBox.

To insert new items in a CheckedListBox, call Add() for each item or use the AddRange() method and send in an array of objects (strings, to be exact) that represent the full set of checkable items. Here is the configuration code (be sure to check out online help for details about these new properties):

```
// Configure the CheckedListBox.
        checkedBoxRadioOptions.Location = new System.Drawing.Point (16, 48);
        checkedBoxRadioOptions.Cursor = Cursors.Hand;
        checkedBoxRadioOptions.Size = new System.Drawing.Size (256, 64);
         checkedBoxRadioOptions.CheckOnClick = true;

        // Add items to the CheckedListBox. checkedBoxRadioOptions.Items.AddRange
        (new object[6] { "Front Speakers", "8-Track Tape Player","CD Player", "Cassette
        Player", "Rear Speakers", "Ultra Base Thumper"} );
```

```
// As always, add the new widget to the Controls collection.
this.Controls.Add (this.checkedBoxRadioOptions);
```

Now update the logic behind the Click event for the Order Button. Ask the CheckedListBox which of its items are currently selected and add them to the orderInfo string. Here are the relevant code updates:

```
protected void btnOrder_Click (object sender, System.EventArgs e)

{

// Build a string to display information.
string orderInfo = "";

...

// For each item in the CheckedListBox:
for(int i = 0; i < checkedBoxRadioOptions.Items.Count; i++)

{

// Is the current item checked?
if(checkedBoxRadioOptions.GetItemChecked(i))
{

// Get text of checked item and append to orderinfo string.
    orderInfo += "Radio Item: ";

    orderInfo +=     checkedBoxRadioOptions.Items[i].ToString();

    orderInfo += "\n";
}}}
```

The final note regarding the CheckedListBox type is that it supports the use of multiple columns through the inherited MultiColumn property. Thus, if you make the following update:

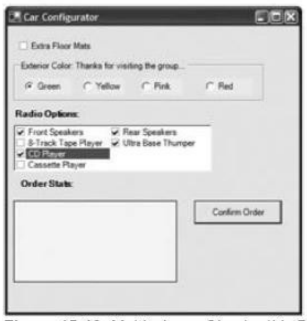


Figure 15-10: Multicolumn CheckedListBox type

```
checkedBoxRadioOptions.MultiColumn =  true;
checkedBoxRadioOptions.ColumnWidth = 130;
```

You see the multicolumn CheckedListBox shown in below Figure.

Important Properties

Property	Description
Appearance	This property is used to set a value determining the appearance of the RadioButton.

AutoCheck	This property is used to set a value indicating whether the Checked value and the appearance of the RadioButton control automatically change when the RadioButton control is clicked.
AutoSize	This property is used to set a value that indicates whether the RadioButton control resizes based on its contents.
BackColor	This property is used to set the background color of the RadioButton control.
CheckAlign	This property is used to set the location of the check box portion of the RadioButton.
Checked	This property is used to set a value indicating whether the RadioButton control is checked.
Font	This property is used to set the font of the text displayed by the RadioButton control.
ForeColor	This property is used to set the foreground color of the RadioButton control.
Location	This property is used to sets the coordinates of the upper-left corner of the RadioButton control relative to the upper-left corner of its form.
Name	This property is used to sets the name of the RadioButton control.
Padding	This property is used to sets padding within the RadioButton control.
Text	This property is used to set the text associated with this RadioButton control.
TextAlign	This property is used to set the alignment of the text on the RadioButton control.
Visible	This property is used to set a value indicating whether the RadioButton control and all its child controls are displayed.

Important Events

Event	Description
Click	This event occurs when the RadioButton control is clicked.
CheckedChanged	This event occurs when the value of the Checked property changes.
AppearanceChanged	This event occurs when the Appearance property value changes.
DoubleClick	This event occurs when the user double-clicks the RadioButton control.
Leave	This event occurs when the input focus leaves the RadioButton control.
MouseClick	This event occurs when the RadioButton control is clicked by the mouse.
MouseDoubleClick	This event occurs when the user double-clicks the RadioButton control with the mouse.
MouseHover	This event occurs when the mouse pointer rests on the RadioButton control.
MouseLeave	This event occurs when the mouse pointer leaves the RadioButton control.

ListBoxes

list box control contains a simple list from which the user can generally select one or more items. List boxes provide limited flexibility compared with List View controls.

List box items can be represented by text strings, bitmaps, or both. If the list box is not large enough to display all the list box items at once, the list box provides a scroll bar. The user scrolls through the list box items and applies or removes selection status as necessary. Selecting a list box item changes its visual appearance, usually by changing the text and background colors to those specified by the relevant operating system metrics. When the user selects or deselects an item, the system sends a notification message to the parent window of the list box.

For an ANSI application, the system converts the text in a list box to Unicode by using the **CP_ACP** code page. This can cause problems. For example, accented Roman characters in a non-Unicode list box in Windows, Japanese version will come out garbled. To fix this, either compile

Creating a List Box

The easiest way to create a list box in a dialog box is to drag it from the Toolbox in Microsoft Visual Studio onto your dialog resource. To create a list box dynamically, or to create a list box in a window other than a dialog box, use the CreateWindowEx function, specifying the WC_LISTBOX window class and the appropriate list box styles.

List Box Types and Styles

There are two types of list boxes: single-selection (the default) and multiple-selection. In a single-selection list box, the user can select only one item at a time. In a multiple-selection list box, the user can select more than one item at a time. To create a multiple-selection list box, specify the LBS_MULTIPLESEL or the LBS_EXTENDEDSEL style.

The appearance and operation of a list box is controlled by list box styles and window styles. These styles indicate whether the list is sorted, arranged in multiple columns, drawn by the application, and so on. The dimensions and styles of a list box are typically defined in a dialog box template that is included in an application's resources.

Note

To use visual styles with these controls, an application must include a manifest and must call InitCommonControls at the beginning of the program. For information on visual styles, see Visual Styles. For information on manifests, see Enabling Visual Styles.

List Box Functions

The DlgDirList function replaces the contents of a list box with the names of drives, directories, and files that match a specified set of criteria. The DlgDirSelectEx function retrieves the current selection in a list box that is initialized by **DlgDirList**. These functions make it possible for the user to select a drive, directory, or file from a list box without typing the location and name of the file.

Also, the GetListBoxInfo function returns the number of items per column in a specified list box.

Notification Messages from List Boxes

When an event occurs in a list box, the list box sends a notification code, in the form of a WM_COMMAND message, to the dialog box procedure of the owner window. List box notification codes are sent when a user selects, double-clicks, or cancels a list box item; when the list box receives or loses the keyboard focus; and when the system cannot allocate enough memory for a list box request. A **WM_COMMAND** message contains the list box identifier in the low-order word of the *wParam* parameter, and the notification code in the high-order word. The *lParam* parameter contains the control window handle.

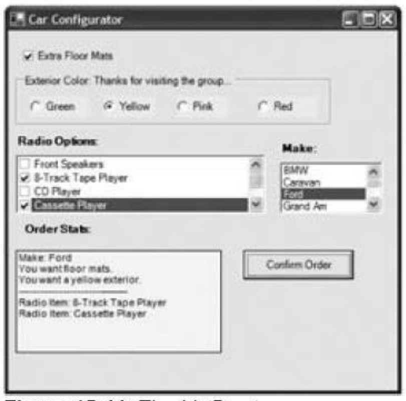
A dialog box procedure is not required to process these messages; the default window procedure processes them.

An application should monitor and process the following list box notification codes.

Notification Messages from List Boxes

Notification code	**Description**
LBN_DBLCLK	The user double-clicks an item in the list box.
LBN_ERRSPACE	The list box cannot allocate enough memory to fulfill a request.
LBN_KILLFOCUS	The list box loses the keyboard focus.
LBN_SELCANCEL	The user cancels the selection of an item in the list box.
LBN_SELCHANGE	The selection in a list box is about to change.
LBN_SETFOCUS	The list box receives the keyboard focus.

ListBoxes and ComboBoxes : As mentioned, the CheckedListBox type inherits most of its functionality from the ListBox type. To illustrate using the ListBox type, let's add another feature to the current CarConfig application: the ability to select the make (BMW, Yugo, and so on) of the automobile. Figure shows the desired UI.



Make: Ford
You want floor mats.
You want a yellow exterior.

Radio Item: 8-Track Tape Player
Radio Item: Cassette Player

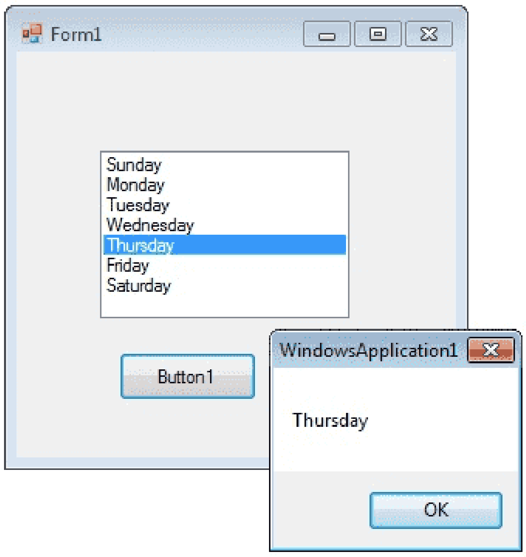
Figure 15-11: The ListBox type

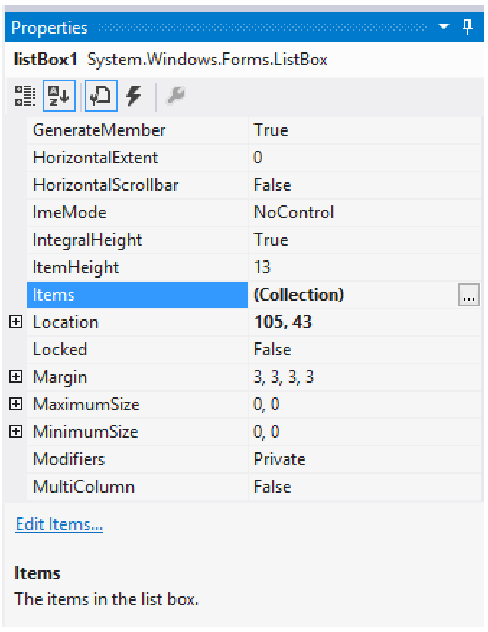
As always, begin by creating a member variable to manipulate your type (in this case a ListBox type). Next, configure the look and feel and insert the new widget in the Form's Controls collection, as shown here:

```
// Configure the list box.
carMakeList.Location = new System.Drawing.Point (168, 48);
 carMakeList.Size = new System.Drawing.Size (112, 67);
carMakeList.BorderStyle = System.Windows.Forms.BorderStyle.FixedSingle;
carMakeList.ScrollAlwaysVisible=  true; carMakeList.Sorted = true;

// Populate the listBox using the AddRange() method.
carMakeList.Items.AddRange(new object[9] { "BMW", "Caravan",
"Ford", "Grand Am", "Jeep", "Jetta", "Saab", "Viper", "Yugo"});

// Add new widget to Form's Control collection.
this.Controls.Add (this.carMakeList);
```

The update to the btnOrder_Click() event handler is also simple, as shown here:

```
protected void btnOrder_Click (object sender, System.EventArgs e)
{

// Get the currently selected item (not index of the item).
if(carMakeList.SelectedItem != null)
orderInfo += "Make: " + carMakeList.SelectedItem + "\n";
...
}
```




Properties

listBox1 System.Windows.Forms.ListBox

GenerateMember	True
HorizontalExtent	0
HorizontalScrollbar	False
ImeMode	NoControl
IntegralHeight	True
ItemHeight	13
Items	(Collection)
⊞ Location	105, 43
Locked	False
⊞ Margin	3, 3, 3, 3
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
Modifiers	Private
MultiColumn	False

Edit Items...

Items
The items in the list box.

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }
    private void Form1_Load(object sender, EventArgs e)
    {
      listBox1.Items.Add("Sunday");
      listBox1.Items.Add("Monday");
      listBox1.Items.Add("Tuesday");
      listBox1.Items.Add("Wednesday");
      listBox1.Items.Add("Thursday");
      listBox1.Items.Add("Friday");
      listBox1.Items.Add("Saturday");
      listBox1.SelectionMode = SelectionMode.MultiSimple;
    }
    private void button1_Click(object sender, EventArgs e)
    {
      foreach (Object obj in listBox1.SelectedItems )
      {
        MessageBox.Show(obj.ToString ());
      }
    }
  }
}
```

ComboBoxes :

Like a ListBox, a ComboBox allows the user to make a selection from a well-defined set of possibilities. However, the ComboBox type is unique in that the user can also insert additional items. Recall that ComboBox derives from ListBox (which then derives from Control). To illustrate its use, add yet another GUI widget to the CarConfig application that allows a user to enter the name of a preferred salesperson.

If the salesperson in question is not on the list, the

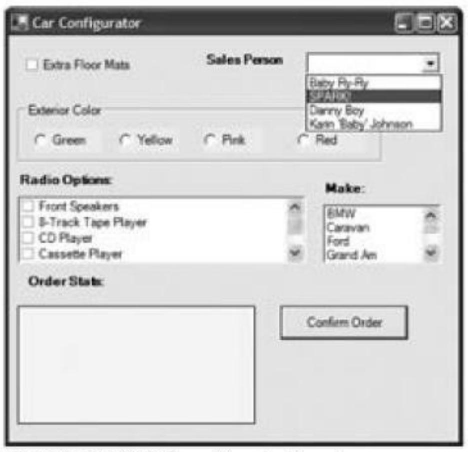


Figure 15-12: The ComboBox type

user can enter a custom name. The GUI update is shown in Figure.

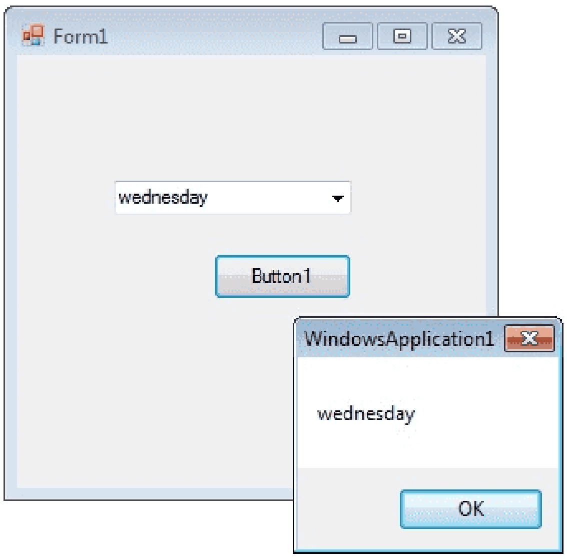
*T*his modification begins with configuring the ComboBox itself. As you can see here, the logic looks identical to that for the ListBox:

```
// ComboBox configuration.
comboSalesPerson.Location = new System.Drawing.Point (152, 16);
comboSalesPerson.Size = new System.Drawing.Size (128, 21);
comboSalesPerson.Items.AddRange(new object[4]{ "Baby Ry-Ry", "SPARK!", "Danny
Boy", "Karin 'Baby' Johnson"});
this.Controls.Add (this.comboSalesPerson);
```

The update to the btnOrder_Click() event handler is again simple, as shown here:

```
protected void btnOrder_Click (object sender, System.EventArgs e)
```

```
{

if(comboSalesPerson.Text != "")
    orderInfo += "Sales Person: " + comboSalesPerson.Text + "\n";
else
    orderInfo += "You did not select a sales person!" + "\n";

}
```



How add a item to combobox

```
comboBox1.Items.Add("Sunday");
comboBox1.Items.Add("Monday");
comboBox1.Items.Add("Tuesday");
```

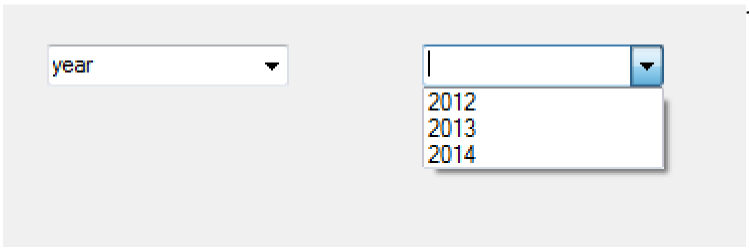
ComboBox SelectedItem

How to retrieve value from ComboBox

If you want to retri

```
string var;
var = comboBox1.Text;
                Or
var item = this.comboBox1.GetItemText(this.comboBox1.SelectedItem);
MessageBox.Show(item);
```

```
using System;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            comboBox1.Items.Add("weekdays");
            comboBox1.Items.Add("year");
        }
        private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            comboBox2.Items.Clear();
            if (comboBox1.SelectedItem == "weekdays")
            {
                comboBox2.Items.Add("Sunday");
                comboBox2.Items.Add("Monday");
                comboBox2.Items.Add("Tuesday");
            }
            else if (comboBox1.SelectedItem == "year")
            {
                comboBox2.Items.Add("2012");
                comboBox2.Items.Add("2013");
                comboBox2.Items.Add("2014");
            }
    }
  }
}
```



Combo Box Types and Styles

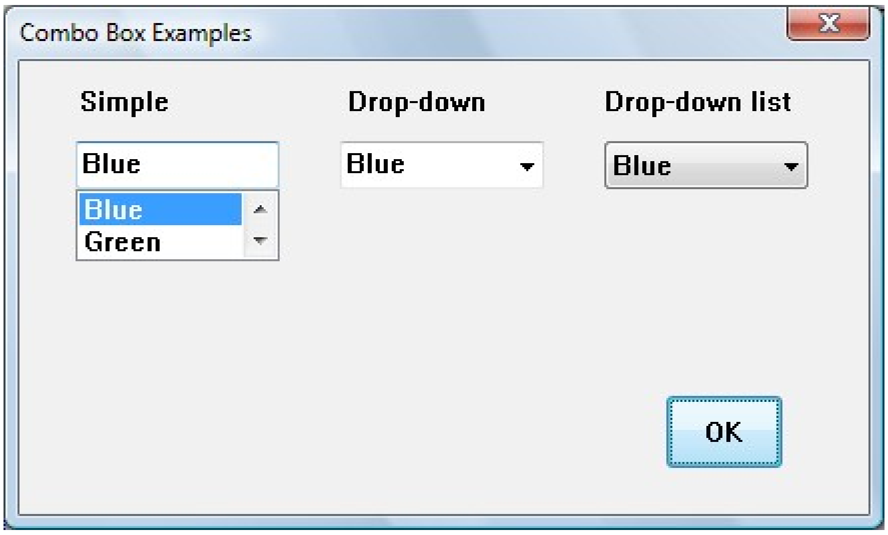
A combo box consists of a list and a selection field. The list presents the options that a user can select, and the selection field displays the current selection. If the selection field is an edit control, the user can enter information not available in the list; otherwise, the user can only select items in the list.

The common controls library includes three main styles of combo box, as shown in the following table.

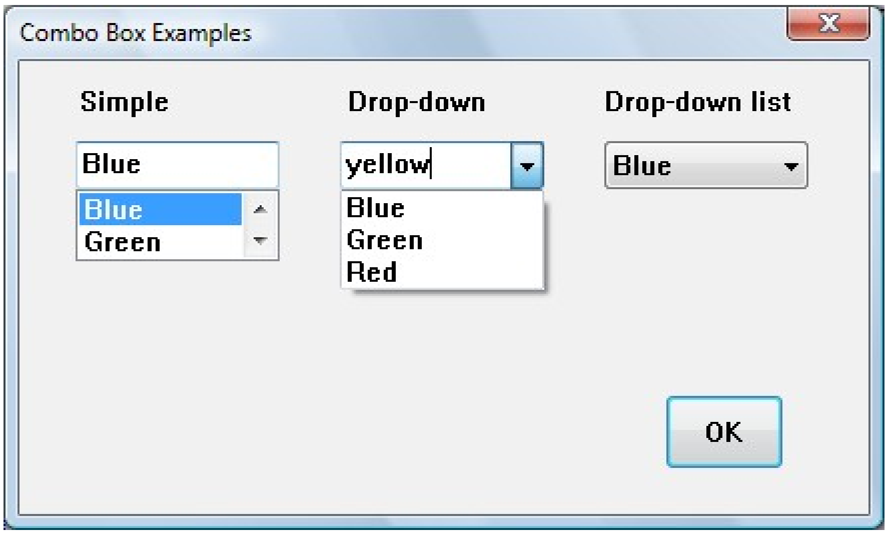
Combo Box Types and Styles

Combo box type	**Style constant**	**Description**
Simple	**CBS_SIMPLE**	Displays the list at all times, and shows the selected item in an edit control.
Drop-down	**CBS_DROPDOWN**	Displays the list when the icon is clicked, and shows the selected item in an edit control.
Drop-down list (drop list)	**CBS_DROPDOWNLIST**	Displays the list when the icon is clicked, and shows the selected item in a static control.

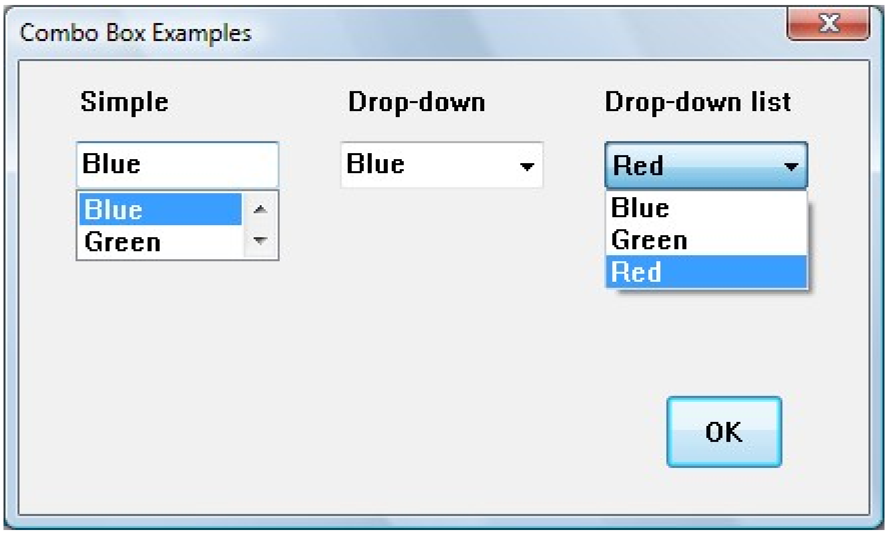
The following screen shots each show the three kinds of combo box as they might appear in Windows Vista. In the first screen shot, the user has selected an item in the simple combo box. The user can also type a new value in the edit box of this control. The list has been sized in the Microsoft Visual Studio resource editor and is only large enough to accommodate two items.



In the second screen shot, the user has typed new text in the edit control of the drop-down combo box. The user could also have selected an existing item. The list box expands to accommodate as many items as possible.



In the third screen shot, the user has opened the drop-down list combo box. The list box expands to accommodate as many items as possible. The user cannot enter new text.



There are also a number of combo box styles that define specific properties. Combo box styles define specific properties of a combo box. You can combine styles; however, some styles apply only to certain combo box types. For a table of combo box styles, see Combo Box Styles.

Note

To use visual styles with combo boxes, an application must include a manifest and must call **InitCommonControls** at the beginning of the program. For information on visual styles, see Visual Styles. For information on manifests, see Enabling Visual Styles.

Combo Box List

The list is the portion of a combo box that displays the items a user can select. Typically, an application initializes the contents of the list when it creates a combo box. Any list item selected by the user is the *current selection*. Multiple items cannot be selected. In simple and drop-down combo boxes, the user can type in the selection field instead of selecting a list item. In these cases, there is no current selection, and it is the application's responsibility to add the item to the list and make it the current selection, if it is appropriate to do so.

This section discusses following topics:

- Current Selection
- Drop-down Lists
- List Contents

Current Selection

The current selection is a list item that the user has selected; the selected text appears in the selection field of the combo box. However, in the case of a simple combo box or a drop-down combo box, the current selection is only one form of possible user input in a combo box. The user can also type text in the selection field.

The current selection is identified by the zero-based index of the selected list item. An application can set and retrieve it at any time. The parent window or dialog box procedure receives notification when the user changes the current selection for a combo box. The parent window or dialog box is not notified when the application changes the selection.

When a combo box is created, there is no current selection. This is also true for a simple or drop-down combo box, if the user has edited the contents of the selection field. To set the current selection, an application sends the **CB_SETCURSEL** message to the combo box. An application can also use the **CB_SELECTSTRING** message to set the current selection to a list item whose string begins with a specified string. To determine the current selection, an application sends the **CB_GETCURSEL** message to the combo box. If there is no current selection, this message returns CB_ERR.

When the user changes the current selection in a combo box, the parent window or dialog-box procedure receives a **WM_COMMAND** message with the CBN_SELCHANGE notification code in the high-order word of the *wParam* parameter. This notification code is not sent when the current selection is set using the **CB_SETCURSEL** message.

A drop-down combo box or drop-down list box sends the CBN_CLOSEUP notification code to the parent window or dialog-box procedure when the drop-down list closes. If the user changed the current selection, the combo box also sends the CBN_SELCHANGE notification code when the drop-down list closes. To execute a specific process each time the user selects a list item, you can handle either the CBN_SELCHANGE or CBN_CLOSEUP notification code. Typically, you would wait for the CBN_CLOSEUP notification code before processing a change in the current selection. This can be particularly important if a significant amount of processing is required.

An application could also process the CBN_SELENDOK and CBN_SELENDCANCEL notification codes. The system sends CBN_SELENDOK when the user selects a list item, or selects an item and then closes the list. This indicates that the user has finished, and that the selection should be processed. CBN_SELENDCANCEL is sent when the user selects an item, but then selects another control, presses ESC while the drop-down list is open, or closes the dialog box. This indicates that the user's selection should be ignored. CBN_SELENDOK is sent before every CBN_SELCHANGE message.

In a simple combo box, the system sends the CBN_DBLCLK notification code when the user double-clicks a list item. In a drop-down combo box or drop-down list, a single click hides the list, so it is not possible to double-click an item.

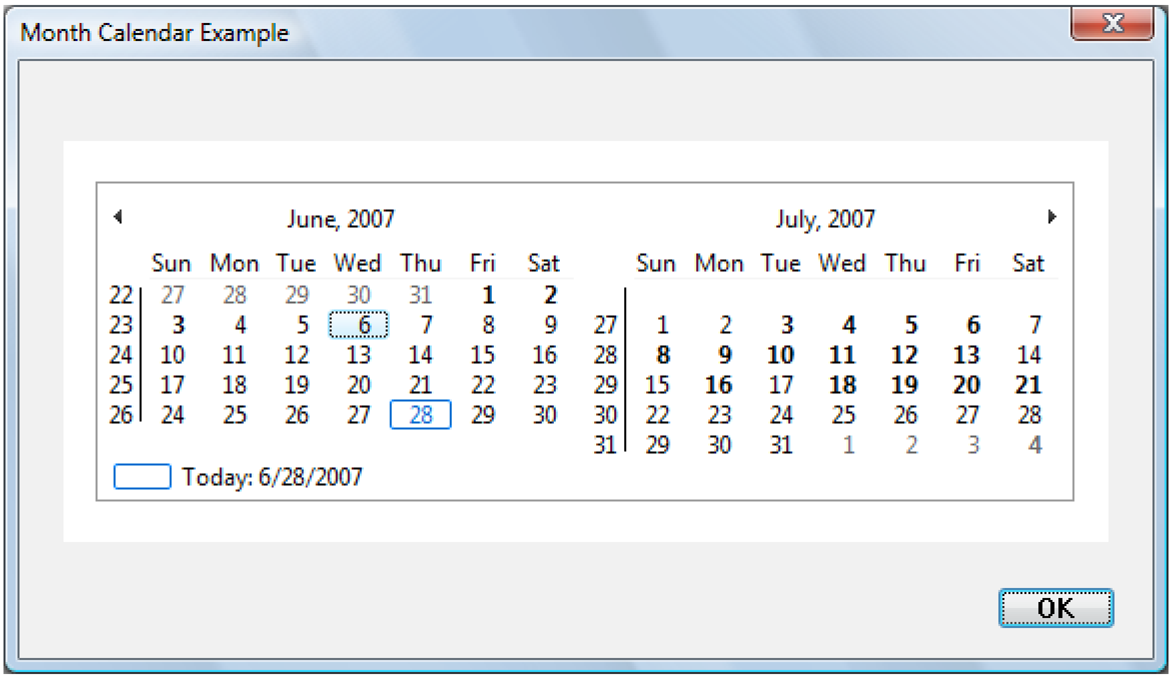
Drop-down Lists

Certain notifications and messages apply only to combo boxes containing drop-down lists. When a drop-down list is open or closed, the parent window of a combo box receives a notification in the form of a **WM_COMMAND** message. If the list is being opened, the high-order word of *wParam* is CBN_DROPDOWN. If the list is being closed, it is CBN_CLOSEUP.

An application can open the list of a drop-down combo box or drop-down list box by using the **CB_SHOWDROPDOWN** message. It can determine whether the list is open by using the **CB_GETDROPPEDSTATE** message and can determine the coordinates of a drop-down list by using the **CB_GETDROPPEDCONTROLRECT** message. An application can also increase the width of a drop-down list by using the **CB_SETDROPPEDWIDTH** message.

Month Calendar Control Features

The following screen shot shows a month calendar control that has been sized to show two months.



June, 2007

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
22	27	28	29	30	31	**1**	**2**
23	**3**	4	5	6	7	8	9
24	10	11	12	13	14	15	16
25	17	18	19	20	21	22	23
26	24	25	26	27	28	29	30

July, 2007

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	1	2	**3**	**4**	**5**	**6**	7
28	**8**	**9**	**10**	**11**	**12**	**13**	14
29	15	**16**	17	**18**	**19**	**20**	**21**
30	22	23	24	25	26	27	28
31	29	30	31	1	2	3	4

Today: 6/28/2007

Note:The appearance and behavior of the month calendar control differs slightly under different versions of the run-time library.

The control in the illustration has the following optional features.

- The current date is shown on a separate line at the bottom of the control. This is the default style.
- The "today circle" (actually a rectangle in this version) appears around the current day, and beside the "Today" line as a visual cue. This is the default style.
- Week numbers are shown at the left of each row of days. This style must be specified.
- Some dates are shown in bold, according to the day state set by the application. For example, dates that have scheduled meetings might be shown in bold. This style must be specified.

Note

Windows does not support dates prior to 1601. See FILETIME for details.

The month-calendar control is based on the Gregorian calendar, which was introduced in 1753. It will not calculate dates that are consistent with the Julian calendar that was in use prior to 1753.

Selecting a day

By default, when a user clicks the arrow buttons in the top left or top right of the month calendar control, the control updates its display to show the previous or next month. The user can also perform the same action by clicking the partial months displayed before the first month and after the last month.

The following keyboard commands can also be used to move the selection. The calendar always scrolls as necessary to display the selected day. (The virtual key codes are shown in the table.)

Selecting a day

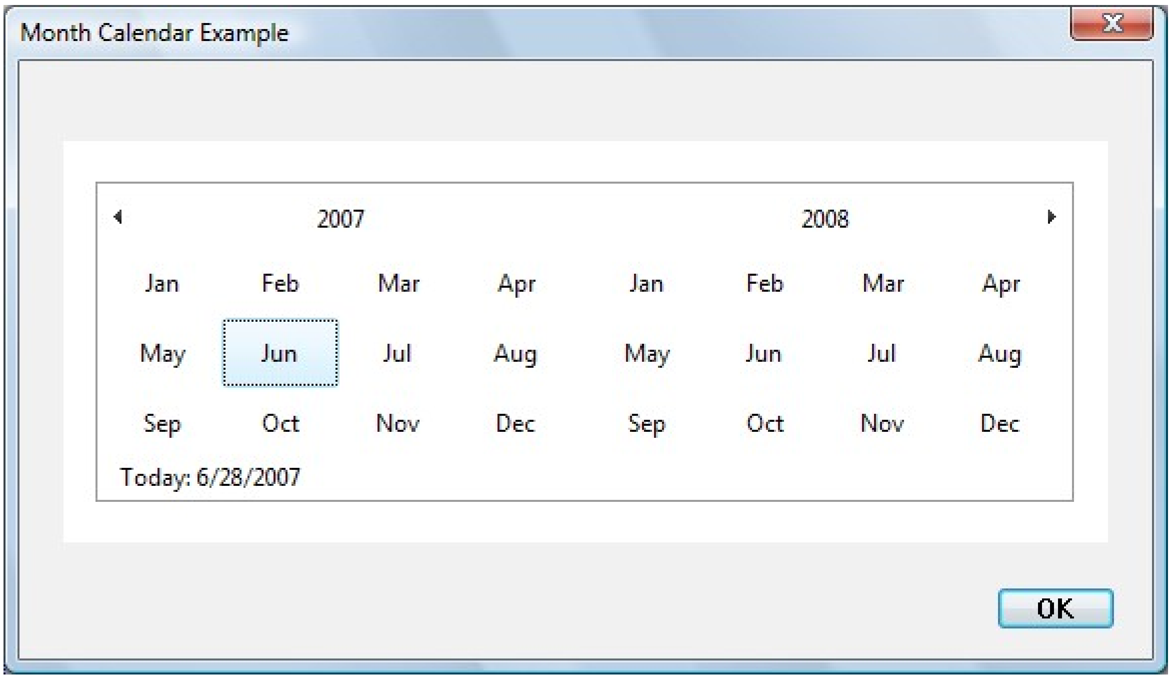
Left arrow (VK_LEFT)	Select the previous day.
Right arrow (VK_RIGHT)	Select the next day.
Up arrow (VK_UP)	Select the same day in the previous week.
Down arrow (VK_DOWN)	Select the same day in the next week.
PAGE UP (VK_PRIOR)	Select the same day in the previous month. (If that month does not have the day, the closest day is selected; for example, the selection moves from March 31 to February 28 or 29.)
PAGE DOWN (VK_NEXT)	Select the same day in the next month.
HOME (VK_HOME)	Select the first day of the current month.
END (VK_END)	Select the last day of the current month.
CTRL + HOME	Scroll one month backward and select a day in the leftmost column.
CTRL + END	Scroll one month forward and select a day in the rightmost column.
CTRL + PAGE UP	Select the same day in an earlier month. The number of months by which the selection moves is the number of months displayed in the control. For example, if two months are displayed, the selection would move from June 6 to May 6.
CTRL + PAGE DOWN	Select the same day in an earlier month. The number of months by which the selection moves is the number of months displayed in the control. For example, if two months are displayed, the selection would move from June 6 to August 6.

If a month calendar control is not using the MCS_NOTODAY style, the user can return to the current day by clicking the "Today" text at the bottom of the control. If the current day is not visible, the control updates its display to show it.

An application can change the number of months by which the control updates its display by using the MCM_SETMONTHDELTA message or the corresponding macro, MonthCal_SetMonthDelta. However, the PAGE UP and PAGE DOWN keys change the selected month by one, regardless of the number of months displayed or the value set by **MCM_SETMONTHDELTA**.

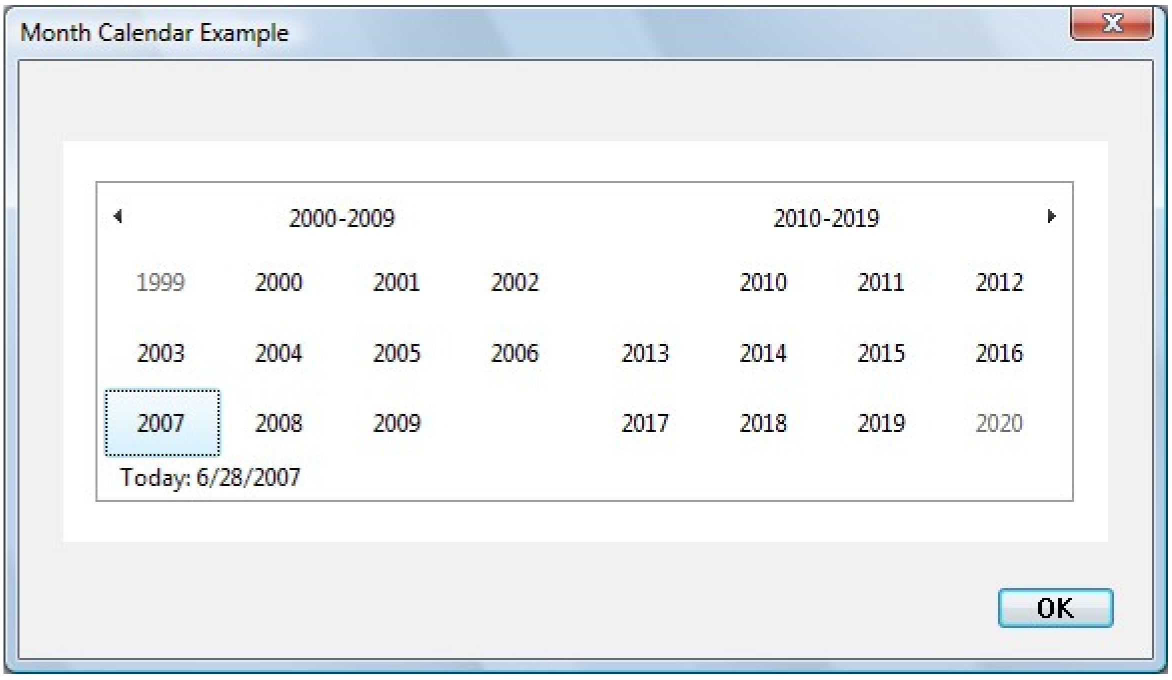
Selecting a nonadjacent month

When a user clicks the name of a displayed month, all months in the year are listed (in earlier versions, this is a pop-up menu). The user can select a month on the list. If the user's selection is not visible, the month calendar control scrolls its display to show the chosen month. In the following screen shot, a month calendar control shows the months of two adjacent years.


Month Calendar Example
2007
2008
Jan Feb Mar Apr Jan Feb Mar Apr
May Jun Jul Aug May Jun Jul Aug
Sep Oct Nov Dec Sep Oct Nov Dec
Today: 6/28/2007
OK

Selecting a different year

If the user clicks the year, a group of years is listed, and the user can select a different one, as shown in the following screen shot.



Localization

The month-calendar control gets its format and all strings from LOCALE_USER_DEFAULT.

Times in the Month Calendar Control

The month calendar control does not display the time. However, the SYSTEMTIME structure that is used to set and retrieve the selected date or today's date contains time fields. When a date is set programmatically, the control either copies the time fields as they are or validates them first and then, if they are invalid, stores the current default times. Following is a list of the messages that set a date and a description of how the time fields are treated.

Times in the Month Calendar Control

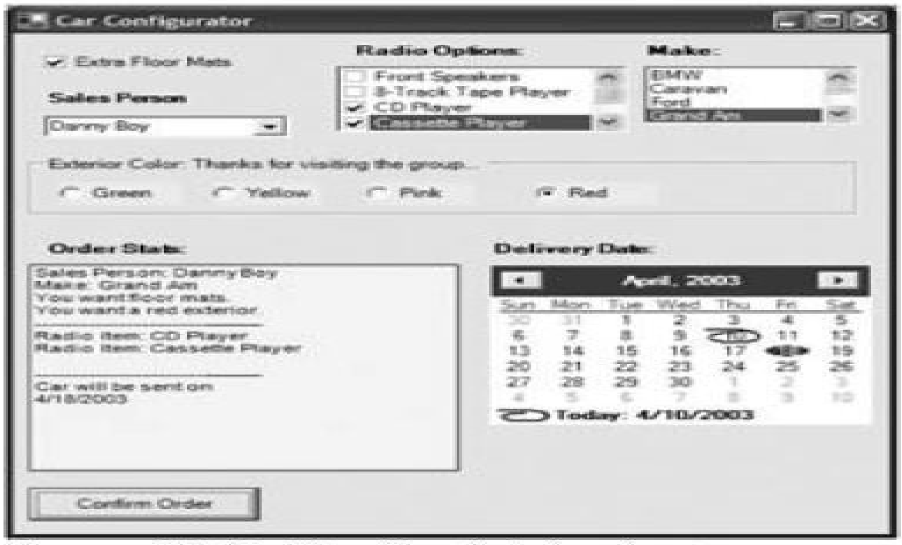
MCM_SETCURSEL	The control copies the time fields as they are, without validation or modification.
MCM_SETRANGE	The time fields of the structures passed in are validated. If they are valid, the time fields are copied without modification. If they are invalid, the control copies the time fields from today's data.
MCM_SETSELRANGE	The time fields of the structures passed in are validated. If they are valid, the time fields are copied without modification. If they are invalid, the control retains the time fields from the current selection ranges.
MCM_SETTODAY	The control copies the time fields as they are, without validation or modification.

When a date is retrieved from the control, the time fields will be copied from the stored times without modification. Handling of the time fields by the control is provided as a convenience to the programmer. The control does not examine or modify the time fields as a result of any operation other than those listed above.

The MonthCalendar Control : The System.Windows.Forms namespace provides an extremely useful widget that allows the user to select a date (or range of dates) using a friendly user interface: the MonthCalendar control. To showcase this new control, update the existing CarConfig application to allow the user to enter in the new vehicle's delivery date. Figure shows the updated (and slightly rearranged) Form.

To begin understanding this new type, examine the core MonthCalendar properties described in Table

```
protected void btnOrder_Click (object sender, System.EventArgs e)
```



Car Configurator

[✓] Extra Floor Mats

Sales Person: Danny Boy

Radio Options: Front Speakers, 8-Track Tape Player, [✓ CD Player], [✓ Cassette Player]

Make: BMW, Caravan, Ford, [✓ Grand Am]

Exterior Color: Thanks for visiting the group...: Green, Yellow, Pink, [✓ Red]

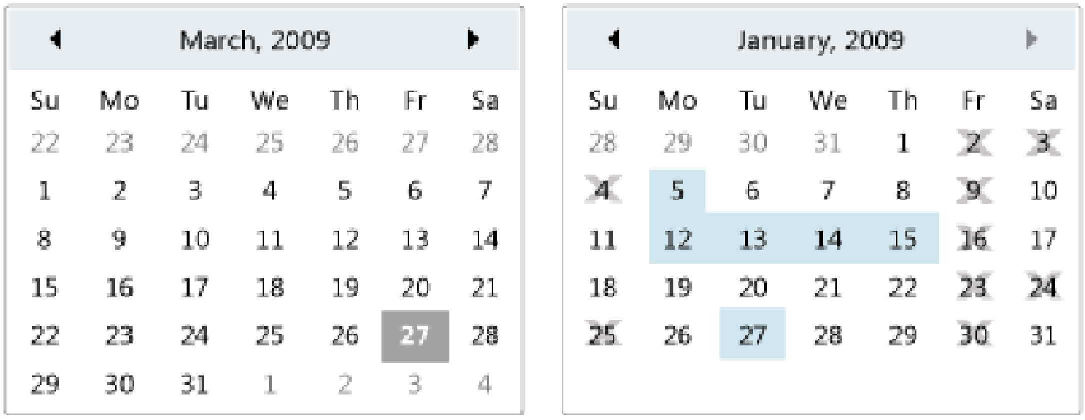
Order Stats:

```
Sales Person: Danny Boy
Make: Grand Am
You want floor mats.
You want a red exterior
_______________________
Radio Item: CD Player
Radio Item: Cassette Player

_______________________
Car will be sent on
4/18/2003
```

Delivery Date: April, 2003 — Today: 4/10/2003

Confirm Order

Figure 15-13: The MonthCalendar type



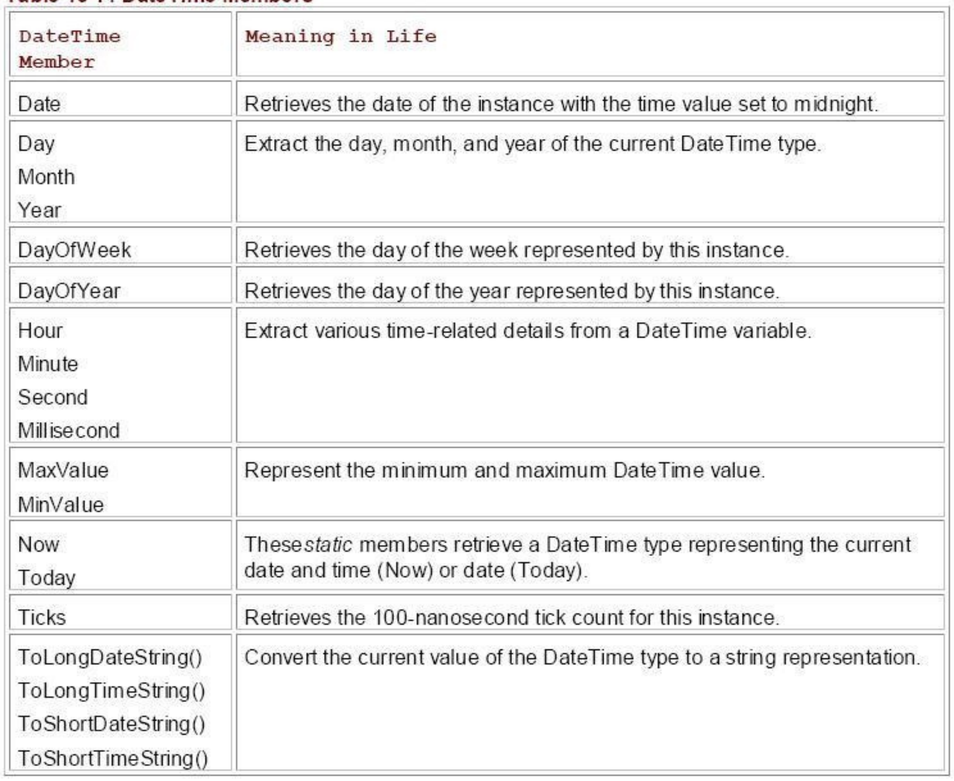
```
Calendar basicCalendar = new Calendar();
basicCalendar.DisplayDateStart = new DateTime(2009, 1, 10);
basicCalendar.DisplayDateEnd = new DateTime(2009, 4, 18);
basicCalendar.DisplayDate = new DateTime(2009, 3, 15);
basicCalendar.SelectedDate = new DateTime(2009, 2, 15);

// root is a Panel that is defined elswhere.
root.Children.Add(basicCalendar);
```

// Get ship date.

```
DateTime d = monthCalendar.SelectionStart;
string dateStr = string.Format("{0}/{1}/{2}", d.Month, d.Day, d.Year);
orderInfo+= "Car will be sent: " + dateStr;
...
}
```

More on the DateTime Type : In the current example, you extracted a DateTime type from the MonthCalendar widget using the SelectionStart and SelectionEnd properties. After this point, you used the Month, Day, and Year properties to build a custom format string. While this is permissible, it is not optimal, given that the DateTime type has a number of built-in formatting options (Table).

Table 15-7: DateTime Members



DateTime Member	Meaning in Life
Date	Retrieves the date of the instance with the time value set to midnight.
Day Month Year	Extract the day, month, and year of the current DateTime type.
DayOfWeek	Retrieves the day of the week represented by this instance.
DayOfYear	Retrieves the day of the year represented by this instance.
Hour Minute Second Millisecond	Extract various time-related details from a DateTime variable.
MaxValue MinValue	Represent the minimum and maximum DateTime value.
Now Today	These*static* members retrieve a DateTime type representing the current date and time (Now) or date (Today).
Ticks	Retrieves the 100-nanosecond tick count for this instance.

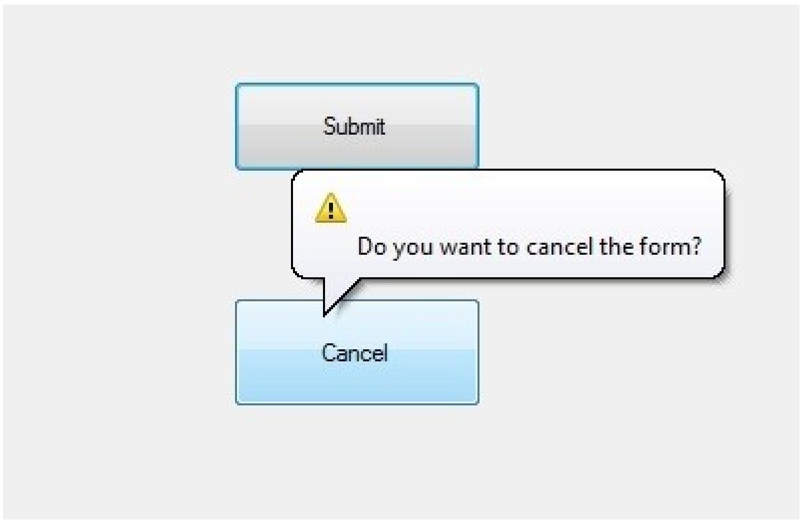
ToLongDateString()
ToLongTimeString()
ToShortDateString()
ToShortTimeString()
	Convert the current value of the DateTime type to a string representation.

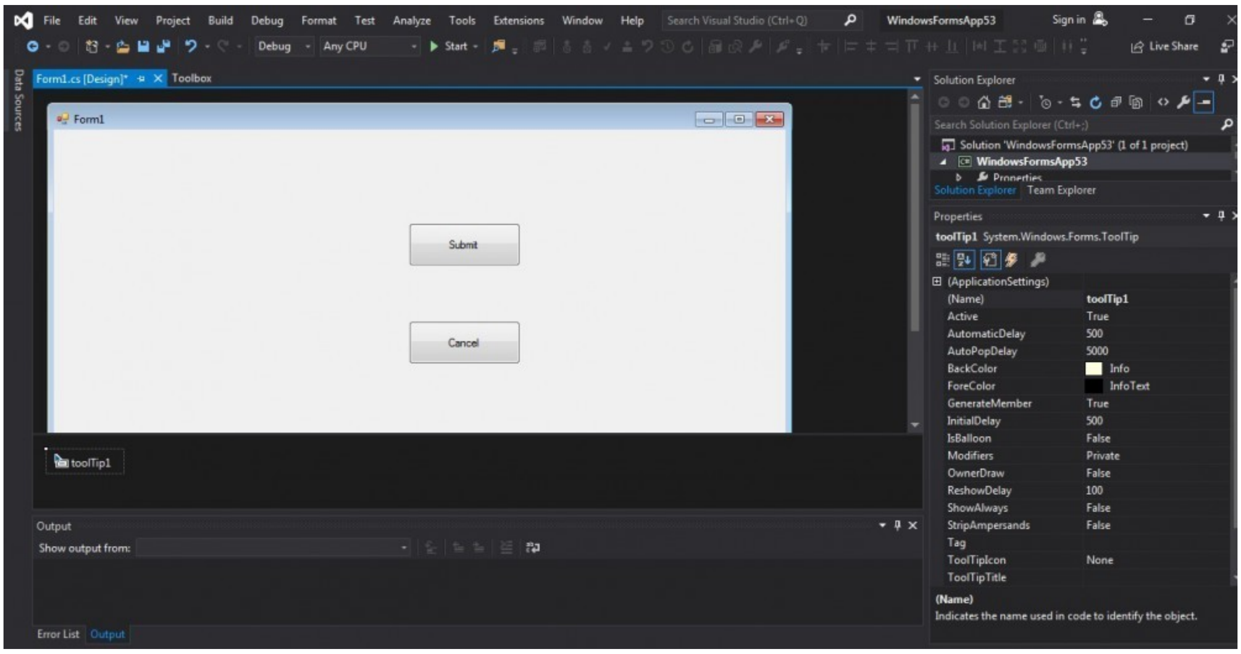
Using these members, you can replace the previous formatting you programmed by hand with the following (you will see no change in the program's output):

string dateStartStr = startD.Date.**ToShortDateString**();
string dateEndStr = endD.Date.**ToShortDateString**();

Assigning ToolTips to Controls :

In the System.Windows.Forms namespace, the ToolTip are simply small floating windows that display a helpful message when the cursor hovers over a given item. Table describes the core members of the ToolTip type.




File Edit View Project Build Debug Format Test Analyze Tools Extensions Window Help
Form1.cs [Design]* Toolbox
Form1
Submit
Cancel
toolTip1
Output
Show output from:
Error List Output
Solution Explorer
Solution 'WindowsFormsApp53' (1 of 1 project)
WindowsFormsApp53

Properties

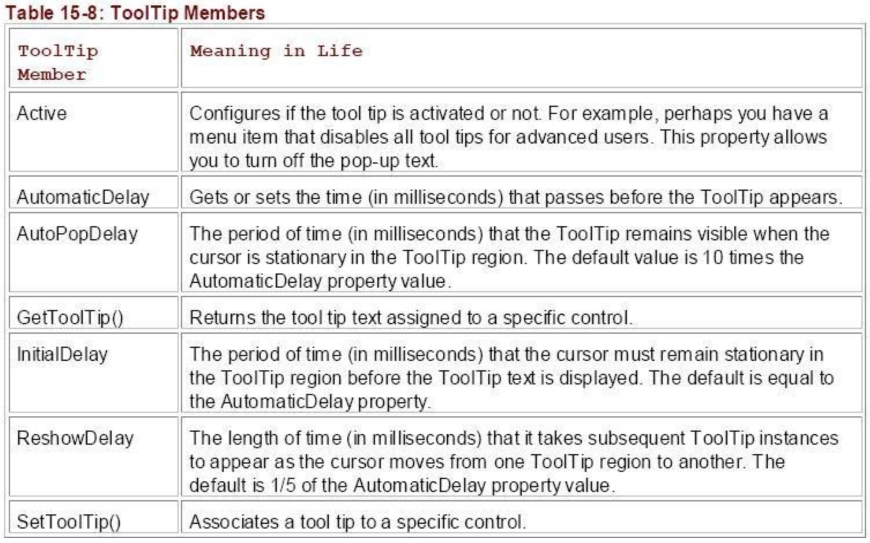
toolTip1 System.Windows.Forms.ToolTip

(ApplicationSettings)	
(Name)	toolTip1
Active	True
AutomaticDelay	500
AutoPopDelay	5000
BackColor	Info
ForeColor	InfoText
GenerateMember	True
InitialDelay	500
IsBalloon	False
Modifiers	Private
OwnerDraw	False
ReshowDelay	100
ShowAlways	False
StripAmpersands	False
Tag	
ToolTipIcon	None
ToolTipTitle	

(Name)
Indicates the name used in code to identify the object.

```
// Create and associate a tool tip to the calendar calendarTip.Active = true;
calendarTip.SetToolTip (monthCalendar,
        "Please select the date (or dates)\n when we can deliver your new car!");
```

Table 15-8: ToolTip Members



ToolTip Member	Meaning in Life
Active	Configures if the tool tip is activated or not. For example, perhaps you have a menu item that disables all tool tips for advanced users. This property allows you to turn off the pop-up text.
AutomaticDelay	Gets or sets the time (in milliseconds) that passes before the ToolTip appears.
AutoPopDelay	The period of time (in milliseconds) that the ToolTip remains visible when the cursor is stationary in the ToolTip region. The default value is 10 times the AutomaticDelay property value.
GetToolTip()	Returns the tool tip text assigned to a specific control.
InitialDelay	The period of time (in milliseconds) that the cursor must remain stationary in the ToolTip region before the ToolTip text is displayed. The default is equal to the AutomaticDelay property.
ReshowDelay	The length of time (in milliseconds) that it takes subsequent ToolTip instances to appear as the cursor moves from one ToolTip region to another. The default is 1/5 of the AutomaticDelay property value.
SetToolTip()	Associates a tool tip to a specific control.

```
// Create and associate a tool tip to the calendar
calendarTip.Active = true;
calendarTip.SetToolTip (monthCalendar,"Please select the date (or date
s)\n when we can deliver your new car!");
```

```
// Creating a ToolTip controlToolTip t_Tip = new ToolTip();
// Seting the properties of ToolTipt_Tip.Active = true;
t_Tip.AutoPopDelay = 4000;
t_Tip.InitialDelay = 600;
 t_Tip.IsBalloon = true;
t_Tip.ToolTipIcon = ToolTipIcon.Info;
 t_Tip.SetToolTip(box1, "Name should start with Capital letter");
 t_Tip.SetToolTip(box2, "Password should be greater than 8 words");
```

Constructor

Constructor	Description
ToolTip()	This Constructors is used to initialize a new instance of the ToolTip without a specified container.
ToolTip(IContainer)	This Constructors is used to initialize a new instance of the ToolTip class with a specified container.

Properties

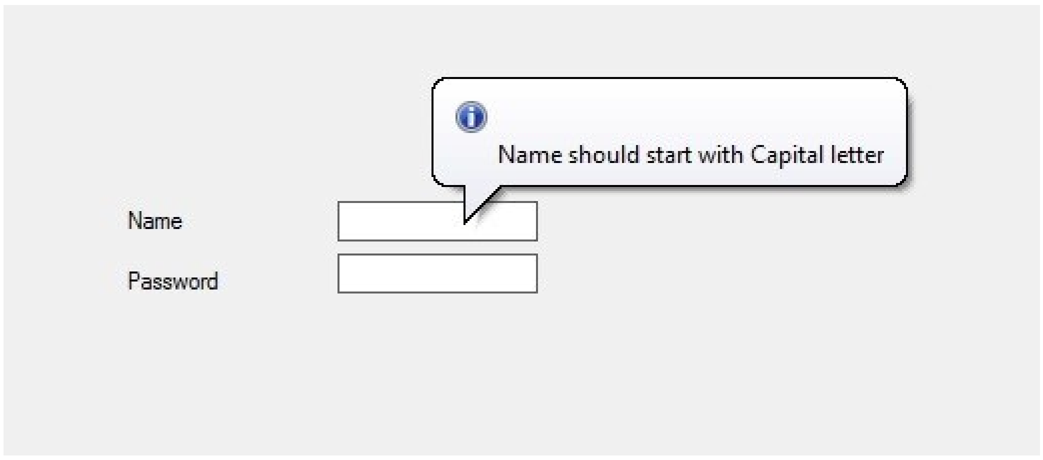
Property	Description
Active	This property is used to get or set a value indicating whether the ToolTip is

	currently active.
AutomaticDelay	This property is used to get or set the automatic delay for the ToolTip.
AutoPopDelay	This property is used to get or set the period of time the ToolTip remains visible if the pointer is stationary on a control with specified ToolTip text.
BackColor	This property is used to get or set the background color for the control.
ForeColor	This property is used to get or set the foreground color of the control.
InitialDelay	This property is used to get or set the time that passes before the ToolTip appears.
IsBalloon	This property is used to get or set a value indicating whether the ToolTip should use a balloon window.
ReshowDelay	This property is used to get or set the length of time that must transpire before subsequent ToolTip windows appear as the pointer moves from one control to another.
ToolTipIcon	This property is used to get or set a value that defines the type of icon to be displayed alongside the ToolTip text.
ToolTipTitle	This property is used to get or set a title for the ToolTip window.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApp34 {

public partial class Form1 : Form {

    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Creating and setting the
        // properties of the Label
        Label l1 = new Label();
        l1.Location = new Point(140, 122);
        l1.Text = "Name";
```

```
// Adding this Label
// control to the form
this.Controls.Add(l1);

// Creating and setting the
// properties of the TextBox
TextBox  box1  =  new TextBox();
box1.Location  =  new Point(248,  119);
box1.BorderStyle  =  BorderStyle.FixedSingle;

// Adding this TextBox
// control to the form
this.Controls.Add(box1);

// Creating and setting the
// properties of Label
Label  l2  =  new Label();
l2.Location  =  new Point(140,  152);
l2.Text  =  "Password";

// Adding this Label
// control to the form
this.Controls.Add(l2);

// Creating and setting the
// properties of the TextBox
TextBox  box2  =  new TextBox();
box2.Location  =  new Point(248,  145);
box2.BorderStyle  =  BorderStyle.FixedSingle;

// Adding this TextBox
// control to the form
this.Controls.Add(box2);

// Creating and setting the
// properties of the ToolTip
ToolTip  t_Tip  =  new ToolTip();
t_Tip.Active  =  true;
t_Tip.AutoPopDelay  =  4000;
t_Tip.InitialDelay  =  600;
t_Tip.IsBalloon  =  true;
t_Tip.ToolTipIcon  =  ToolTipIcon.Info;
t_Tip.SetToolTip(box1,  "Name should start with Capital letter");
t_Tip.SetToolTip(box2,  "Password should be greater than 8 words");
```

```
    }
}
}
```



Tooltip Properties

- Active - A tooltip is currently active.
- AutomaticDelay - Automatic delay for the tooltip.
- AutoPopDelay - The period of time the ToolTip remains visible if the pointer is stationary on a control with specified ToolTip text.
- InitialDelay - Gets or sets the time that passes before the ToolTip appears.
- IsBaloon - Gets or sets a value indicating whether the ToolTip should use a balloon window.
- ReshowDelay - Gets or sets the length of time that must transpire before subsequent ToolTip windows appear as the pointer moves from one control to another.
- ShowAlways - Displays if tooltip is displayed even if the parent control is not active.
- ToolTipIcon - Icon of tooltip window.
- ToolTipTitle - Title of tooltip window.
- UseAnimation - Represents whether an animation effect should be used when displaying the tooltip.
- UseFading - Represents whether a fade effect should be used when displaying the tooltip.

Tooltips appear automatically, or pop up, when the user pauses the mouse pointer over a tool or some other UI element. The tooltip appears near the pointer and disappears when the user clicks a mouse button, moves the pointer away from the tool, or simply waits for a few seconds.

The tooltip control in the following illustration displays information about a file on the Windows desktop. As you move the mouse over the illustration, you should also see a live tooltip that contains descriptive text.

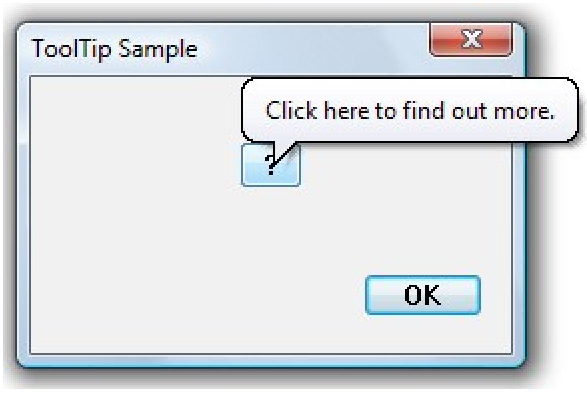
Tooltip Behavior and Appearance

Tooltip controls can display a single line of text or multiple lines. Their corners can be rounded or square. They might or might not have a stem that points to the tools like a cartoon speech balloon. Tooltip text can be stationary or can move with the mouse pointer, called tracking. Stationary text can be displayed adjacent to a tool or it can be displayed over a tool, which is referred to as in-place. Standard tooltips are stationary, display a single line of text, have square corners, and have no stem pointing to the tool.

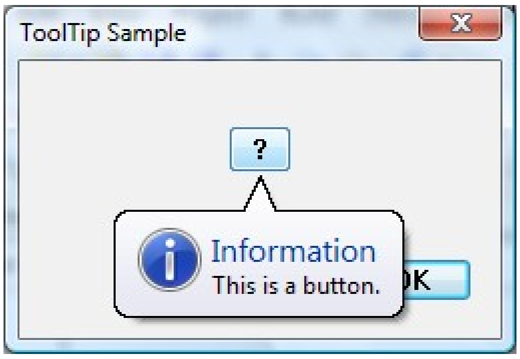
Tracking tooltips, which are supported by version 4.70 of the common controls, change position on the screen dynamically. By rapidly updating the position, these tooltip controls appear to move smoothly, or "track." These are useful when you want tooltip text to follow the position of the mouse pointer as it moves. For more information about tracking tooltips and an example with code that shows how you create them, see Tracking Tooltips.

Multiline tooltips, which are also supported by version 4.70 of the common controls, display text on more than one line. These are useful for displaying lengthy messages. For more information and an example that shows how to create multiline tooltips, see Multiline Tooltips.

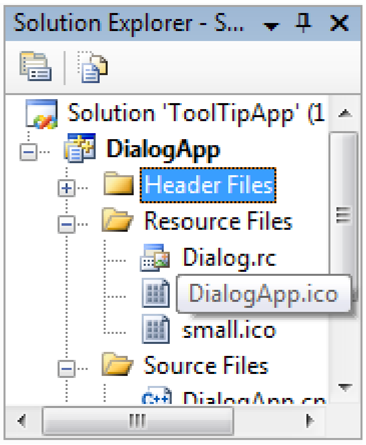
Balloon tooltips are displayed in a box with rounded corners and a stem pointing to the tool. They can be either single-line or multiline. The following illustration shows a balloon tooltip with the stem and rectangle in their default positions. For more information about balloon tooltips and an example that shows how to create them, see Using Tooltip Controls.



A tooltip can also have title text and an icon, as shown in the following illustration. Note that the tooltip must have text; if it has only title text, the tooltip does not display. Also the icon does not appear unless there is a title.



Sometimes text strings are clipped because they are too long to be displayed completely in a small window. In-place tooltips are used to display text strings for objects that have been clipped, such as the file name in the following illustration. For an example that shows how to create in-place tooltips, see In-Place Tooltips.



The cursor must hover over a tool for a period of time before the tooltip is displayed. The default duration of this timeout is controlled by the user's double click time and is typically about one-half second. To specify a non-default timeout value, send the tooltip control a **TTM_SETDELAYTIME** message.

Creating Tooltip Controls

To create a tooltip control, call **CreateWindowEx** and specify the **TOOLTIPS_CLASS** window class. This class is registered when the common control DLL is loaded. To ensure that this DLL is loaded, include the **InitCommonControlsEx** function in your application. You must explicitly define a tooltip control as topmost. Otherwise, it might be covered by the parent window. The following code fragment shows how to create a tooltip control.

```
HWND hwndTip = CreateWindowEx(NULL, TOOLTIPS_CLASS, NULL,
                            WS_POPUP | TTS_NOPREFIX | TTS_ALWAYSTIP,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            hwndParent, NULL, hinstMyDll,
                            NULL);

SetWindowPos(hwndTip, HWND_TOPMOST,0, 0, 0, 0,
             SWP_NOMOVE | SWP_NOSIZE | SWP_NOACTIVATE);
```

The window procedure for the tooltip control automatically sets the size, position, and visibility of the control. The height of the tooltip window is based on the height of the font currently selected into the device context for the tooltip control. The width varies based on the length of the string currently in the tooltip window.

Activating Tooltip Controls

A tooltip control can be either active or inactive. When it is active, the tooltip text appears when the mouse pointer is on a tool. When it is inactive, the tooltip text does not appear, even if the pointer is on a tool. The **TTM_ACTIVATE** message activates and deactivates a tooltip control.

Supporting Tools

A tooltip control can support any number of tools. To support a particular tool, you must register the tool with the tooltip control by sending the control the **TTM_ADDTOOL** message. The message includes the address of a **TOOLINFO** structure, which provides information the tooltip control needs to display text for the tool. The **uID** member of the **TOOLINFO** structure is defined by the application. Each time you add a tool, your application provides a unique identifier. The **cbSize** member of the **TOOLINFO** structure is required, and must specify the size of the structure.

A tooltip control supports tools implemented as windows (such as child windows or control windows) and as rectangular areas within a window's client area. When you add a tool implemented as a rectangular area, the **hwnd** member of the **TOOLINFO** structure must specify the handle to the window that contains the area, and the **rect** member must specify the client coordinates of the area's bounding rectangle. In addition, the **uID** member must specify the application-defined identifier for the tool.

When you add a tool implemented as a window, the **uID** member of the **TOOLINFO** structure must contain the window handle to the tool. Also, the **uFlags** member must specify the **TTF_IDISHWND** value, which tells the tooltip control to interpret the **uID** member as a window handle.

Displaying Text

When you add a tool to a tooltip control, the **lpszText** member of the **TOOLINFO** structure must specify the address of the string to display for the tool. After you add a tool, you can change the text using the **TTM_UPDATETIPTEXT** message.

If the high-order word of **lpszText** is zero, the low-order word must be the identifier of a string resource. When the tooltip control needs the text, the system loads the specified string resource from the application instance identified by the **hinst** member of the **TOOLINFO** structure.

If you specify the LPSTR_TEXTCALLBACK value in the **lpszText** member, the tooltip control notifies the window specified in the **hwnd** member of the **TOOLINFO**structure whenever the tooltip control needs to display text for the tool. The tooltip control sends the TTN_GETDISPINFO notification code to the window. The message includes the address of a **NMTTDISPINFO** structure, which contains the window handle as well as the application-defined identifier for the tool. The window examines the structure to determine the tool for which text is needed, and it fills the appropriate structure members with information that the tooltip control needs in order to display the string.

Note:

The maximum length for standard tooltip text is 80 characters. For more information, see the **NMTTDISPINFO** structure. Multiline tooltip text can be longer.

Many applications create toolbars containing tools that correspond to menu commands. For such tools, it is convenient for the tooltip control to display the same text as the corresponding menu item. The system automatically strips the ampersand (&) accelerator characters from all strings passed to a tooltip control, and terminates the string at the first tab character (\t), unless the control has the **TTS_NOPREFIX** style.