System.Windows.Forms

The System.Windows.Forms namespace contains a large number of types to aid in the process of building rich user interfaces.

The classes in this namespace can be grouped into the following categories:
- Control,
- User Control, and
- Form.
The Control Class

Most classes within the System.Windows.Forms namespace derive from the Control class.

The Control class provides the base functionality for all controls that are displayed on a Form. The Form class represents a window within an application. This includes dialog boxes, modeless windows, and Multiple Document Interface (MDI) client and parent windows.

To create a custom control that is a composite of other controls, use the UserControl class.
Controls

- The System.Windows.Forms namespace provides a variety of control classes that allow you to create rich user interfaces.

- Some controls are designed for data entry within the application, such as TextBox and ComboBox controls. Other controls display application data, such as Label and ListView. The namespace also provides controls for invoking commands within the application, such as Button and ToolBar.
Components

The System.Windows.Forms namespace provides also classes that do not derive from the Control class but still provide visual features to a Windows-based application.

For example, ToolTip and ErrorProvider extend the capabilities or provide information to the user. Menu, MenuItem, and ContextMenu provide the ability to display menus, which can be used to invoke commands within an application. The Help and HelpProvider classes enable you to display help information to the user of your applications.
MessageBox

The System.Windows.Forms namespace provides also the MessageBox class for displaying a message box that can display and retrieve data from the user.

This message box is shown when we exit WinTicTacToe.
# Core Windows Form Types

<table>
<thead>
<tr>
<th>Windows Form Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>This class represents heart of a Windows Forms application. Using the methods of Application, one is able to process Windows messages, start and terminate a Windows Form application, etc.</td>
</tr>
<tr>
<td>ButtonBase, Button, CheckBox, ComboBox, DataGrid, GroupBox, ListBox, LinkLabel, PictureBox, Form</td>
<td>These classes (and many others) represent types that corresponds to various GUI widgets. This type represents a main window (or dialog box) of a Windows Form application.</td>
</tr>
</tbody>
</table>
## Core Windows Form Types II

<table>
<thead>
<tr>
<th>Windows Form Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorDialog, FileDialog,</td>
<td>Windows Forms defines a number of canned dialog boxes.</td>
</tr>
<tr>
<td>FontDialog, PrintPreviewDialog</td>
<td>These types are used to build top-most and context-sensitive (pop-up) menu systems.</td>
</tr>
<tr>
<td>Menu, MainMenu, MenuItem,</td>
<td>Various utility types to facilitate interactive GUIs.</td>
</tr>
<tr>
<td>ContextMenu</td>
<td>Various types used to equip a Form with common child controls.</td>
</tr>
<tr>
<td>Clipboard, Help, Timer, Screen,</td>
<td></td>
</tr>
<tr>
<td>ToolTip, Cursors</td>
<td></td>
</tr>
<tr>
<td>StatusBar, Splitter, ToolBar,</td>
<td></td>
</tr>
<tr>
<td>ScrollBar</td>
<td></td>
</tr>
</tbody>
</table>
Building a Main Window

In order to create a new main window, one has to perform 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 the new derived `Form` class as an argument.
namespace FirstWindow
{
    using System;
    using System.Windows.Forms;

    public class SimpleWindow : Form
    {
        public SimpleWindow() { // any user-specific configuration }

        public static void Main( string[] args )
        {
            Application.Run( new SimpleWindow() );
        }
    }
}
Output

C:\> csc /target:winexe SimpleWindow.cs
C:\> SimpleWindow

The Windows Form is displayed using default values.
Adding a Button

```csharp
public class SimpleWindow : Form
{
    public SimpleWindow()
    {
        Button lBtn = new Button(); // create a Button object
        lBtn.Text = "Show Time"; // set Button text
        lBtn.Top = 100; // set position
        lBtn.Left = 100;
        lBtn.Click += new EventHandler(ClickHandler); // register handler
        Controls.Add(lBtn); // add button to Form controls
    }

    private void ClickHandler(object sender, EventArgs e)
    {
        Text = String.Format("Click Time: {0:T}", DateTime.Now);
    }
}
```
Output

C:\> csc /target:winexe SimpleButtonWindow.cs
C:\> SimpleButtonWindow

The Windows Form is displayed using default values.
Overriding Form Methods

```csharp
public class SimpleWindow : Form
{
    public SimpleWindow() { // any user-specific configuration }

    protected override void OnVisibleChanged(EventArgs e)
    {
        base.OnVisibleChanged(e);
        MessageBox.Show( "Thank you for playing with Forms." );
    }

    public static void Main(string[] args)
    {
        Application.Run(new SimpleWindow());
    }
}
```
C:\> csc /target:winexe SimpleWindow.cs
C:\> SimpleWindow

Output

Thank you for playing with Forms.

Show

Hide

Thank you for playing with Forms.
The Application Class

-The Application class is a low-level class that defines members that allow one to control various behaviors of a Windows Forms application.

-The Application class defines also a set of events that allow one to respond to application-level events such as application shutdown and idle processing.
## Application Core Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddMessageFilter()</td>
<td>These methods allow one to intercept messages for any necessary preprocessing. A message filter must implement the IMessageFilter interface.</td>
</tr>
<tr>
<td>RemoveMessageFilter()</td>
<td></td>
</tr>
<tr>
<td>DoEvents()</td>
<td>Provides the ability for an application to process messages currently in the message queue.</td>
</tr>
<tr>
<td>Exit()</td>
<td>Terminates the application.</td>
</tr>
<tr>
<td>ExitThread()</td>
<td>Exists the message loop on the current thread and closes all windows owned by the current thread.</td>
</tr>
<tr>
<td>Run()</td>
<td>Starts a standard message loop on the current thread.</td>
</tr>
</tbody>
</table>
# Application Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationExit</td>
<td>Occurs when the application is just about to shut down.</td>
</tr>
<tr>
<td>Idle</td>
<td>Occurs when the application’s message loop has finished processing and is about to enter an idle state.</td>
</tr>
<tr>
<td>ThreadExit</td>
<td>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.</td>
</tr>
</tbody>
</table>
Responding to ApplicationExit

```csharp
public class MainForm : Form
{
    ...
    public MainForm()
    {
        ...
        // intercept the ApplicationExit event
        Application.ApplicationExit += new EventHandler( OnFormExit );
    }

    // event handler
    private void OnFormExit( object sender, EventArgs e )
    {
        MessageBox.Show( "Thank you for playing with SimpleWindow." );
    }
}
```
Message Preprocessing

using Microsoft.Win32;

public class AMessageFilter : IMessageFilter
{
    public bool PreFilterMessage( ref Message aMessage )
    {
        // intercept the left mouse button down message
        if ( aMessage.Msg == 513 ) // WM_LBUTTONDOWN
        {
            MessageBox.Show( "WM_LBUTTONDOWN is: " + aMessage.Msg );
            return true; // prevent this message from being dispatched
        }
        // allow message to continue to the next filter or control
        return false;
    }
}

Com S 430
public class MainForm : Form
{
    private AMessageFilter fFilter = new AMessageFilter();

    public MainForm()
    {
        Application.AddMessageFilter( fFilter );
        Application.ApplicationExit += new EventHandler( OnFormExit );
    }

    private void OnFormExit( object sender, EventArgs e )
    {
        Application.RemoveMessageFilter( fFilter );
        ...
    }
}
Output

- WM_LBUTTONDOWN is: 513
- Thank you for playing with Forms.
- Thank you for playing with SimpleWindow.
.NET GUI Programming

.NET provides two GUI toolkits “Windows Forms” and “WebForms” to facilitate the development of graphical user interfaces.

The namespace System.Windows.Forms contains a number of types that allow one to build traditional Windows desktop applications. In fact, this namespace hides the raw Win32 primitives allowing one to focus on the functionality of your application.

The WebForms types are contained in the namespaces System.Web.UI and System.Web.UI.WebControls. WebForm types are used to build browser-independent Web applications.
The user interface in .NET (Windows) is event based. If an event occurs the operating system will forward this event (including additional information) to the corresponding application.

While running, an application receives messages from the (Windows) operating system. The application defines a main message loop that will process messages.

In the message loop all incoming messages are checked and, if necessary, an application-defined event handler is called.

The application design has to guarantee that the operating system is not blocked and can deliver messages to the application, respectively.
Case Study - TicTacToe

We implement two versions of TicTacToe:

- A console application that will ask for the names of the two players (the name “Computer” stands for the machine) and the game will in turn ask the first player to make a move, print to board on the screen, check the game status, ask the second player to make a move, print to board on the screen, check the game status.

- A Windows application that allows a user to play against the computer. The user will be able to select how can make a move first.
System Architecture

Model

fBoard

Controller

View

Update State

Update View

Com S 430
public interface ITicTacToeView
{
    void RegisterGame( TicTacToe aGame );
    void PrintBoard();
    void Draw();
    void GameWon( int aPlayer );
}
The Model

public class TicTacToe
{
    public TicTacToe( ITicTacToeView aView )
    {
        // register View
        fView = aView;
        fView.RegisterGame( this ); // connect View and Model
        ...
    }

    public int this [int aIndex] { get { ... } set { ... } } // update state
    public int ComputerMove() { ... } // automatic move
    public bool Play { get { ... } } // manual move
    public void CheckGameOver( int aPlayer ) { ... } // Game over?
}

Com S 430
public TicTacToeView()
{
    new TicTacToe( this ); // the model
    
    // Ask for the names of the players
    for ( int i = 0; i < 2; i++ )
    {
        if ( fPlayers[i].ToUpper().Equals( "COMPUTER" ) )
        {
            fComputerPlayer[i] = true;
        }
        else
        {
            fComputerPlayer[i] = false;
        }
    }

    while ( fModel.Play )
    {
        if ( fComputerPlayer[fPlayerId] )
        {
            ComputerMove();
        }
        else
        {
            PlayerMove();
        }
    }
}
private void PlayerMove()
{
    while (true) {
        Console.Write("Player {0} make a move: ", fPlayers[fPlayerId]);
        try
        {
            // handle user input
            int lNumber = Int32.Parse(Console.ReadLine());
            int lRow = lNumber / 10 - 1;
            int lColumn = lNumber % 10 - 1;
            PerformMove(lRow * 3 + lColumn);
            break;
        }
        catch (Exception e)
        {
            Console.WriteLine("...");
        }
    }
}
private void ComputerMove()
{
    // perform a model method
    int lMove = fModel.ComputerMove();
    int lRow = lMove / 3;
    int lColumn = lMove % 3;
    Console.WriteLine( … );
    PerformMove( lMove );
}
private void PerformMove( int aMove )
{
    int lRow = aMove / 3;
    int lColumn = aMove % 3;
    // update Model state, Model will call back View
    // Controller -> Model -> View
    fModel[aMove] = fPlayerId;
    // update Model state, Model will call back View
    // Controller -> Model -> View
    fModel.CheckGameOver( fPlayerId );
    // update Controller state
    fPlayerId = ++fPlayerId % 2;
}
public void PrintBoard()
{
    Console.WriteLine( "\{0\} \{1\} \{2\}\); // print first row
    Console.WriteLine( "\{0\} \{1\} \{2\}\); // print second row
    Console.WriteLine( "\{0\} \{1\} \{2\}\); // print third row
}

public void Draw()
{
    Console.WriteLine( "Game over!" );
}

public void GameWon( int aPlayer )
{
    Console.WriteLine( "Player \{0\} has won!\", fPlayers[aPlayer] );
}
Output
WinTicTacToe

Configuration Menu

Field (Label) “label_2_1”

Board

Game status
public MainForm()
{
    InitializeComponent();

    fPlayers = new string[2];
    fPlayers[0] = "Computer";
    fPlayers[1] = "You";

    fComputerFristMenuItem_Click( null, null );
    NewGame_Click( null, null );

    // start game
    Show(); // show Form
}

Wait for user input controlled by Win32.
Mouse Click Event Handling

```csharp
private void label_1_1_Click(object sender, System.EventArgs e)
{
    UserMove( 0 ); // position 1,1 selected
}

private void UserMove( int aMove )
{
    try
    {
        PerformMove( aMove );
        ComputerMove(); // computer is always the next player
    }
    catch {} // simply ignore mouse click
}
```

There are 9 event handler. One for every field.
private void ComputerMove()
{
    PerformMove( fModel.ComputerMove() );
}
private void PerformMove( int aMove )
{
    int lRow = aMove / 3;
    int lColumn = aMove % 3;
    // update Model state, Model will call back View
    // Controller -> Model -> View
    fModel[aMove] = fPlayerId;
    // Update game status first!
    fPlayerName.Text = fPlayers[fPlayerId];
    fLastFieldSet.Text = "("+lRow.ToString()+","+lColumn.ToString()+")";
    // update Model state, Model will call back View
    // Controller -> Model -> View
    fModel.CheckGameOver( fPlayerId );
    fPlayerId d = ++fPlayerId d % 2;
}

Win32 - PerformMove()
public void PrintBoard()
{
    PrintField( label_1_1, 0 );
    ...
    PrintField( label_3_3, 8 );
}

public void Draw()
{
    MessageBox.Show( "Game over!" );
}

public void GameWon( int aPlayer )
{
    MessageBox.Show( "…" );
}

Com S 430
Output
The Control Class

- The Control class establishes the common behaviors required by any GUI-centric type.

- The core members of Control allow one to configure the size and position of a control, extract the underlying HWND, as well as capture keyboard and mouse input.
# Control Core Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top, Left, Bottom, Right, Bounds, ClientRectangle, Height, Width</td>
<td>Each of these properties specifies various attributes about the current dimensions of the Control-derived object.</td>
</tr>
<tr>
<td>Created, Disposed, Enabled, Focused, Visible</td>
<td>These properties each return a Boolean that specifies the state of the Control.</td>
</tr>
<tr>
<td>Handle</td>
<td>HWND of the Control</td>
</tr>
<tr>
<td>ModifierKeys</td>
<td>This static property checks the current state of the modifier keys.</td>
</tr>
<tr>
<td>MouseButtons</td>
<td>This static property checks the current state of the mouse buttons.</td>
</tr>
<tr>
<td>Parent</td>
<td>Returns the Control’s parent.</td>
</tr>
<tr>
<td>TabIndex, TabStop</td>
<td>These properties are used to configure the tab order of the Control.</td>
</tr>
<tr>
<td>Text</td>
<td>The current text associated with this Control.</td>
</tr>
</tbody>
</table>
### Core Control Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetStyle(), SetStyle()</td>
<td>Theses methods are used to manipulate the style flags of the Control.</td>
</tr>
<tr>
<td>Hide(), Show(),</td>
<td>These methods indirectly set the state of the Visible property. Invalidate and Redraw force the Control to redraw itself.</td>
</tr>
<tr>
<td>Invalidate(), Redraw()</td>
<td></td>
</tr>
<tr>
<td>OnXXXX()</td>
<td>Event handler that can be overridden by a subclass.</td>
</tr>
<tr>
<td>SetBounds(), SetLocation(),</td>
<td>Each of these methods is used to establish the dimensions of the Control-derived object.</td>
</tr>
<tr>
<td>SetClientArea()</td>
<td></td>
</tr>
</tbody>
</table>

Com S 430
The Color Manager consists of a Color Controller that is used to manipulate a RGB color value and an unlimited number of RGB views.

The Color Manager is a publisher/subscriber application. That is, every time the color value is changed the controller will notify all its subscribers.

The Color Manager uses a multicast delegate to inform all its dependent views.
The Color Manager

216 145 74

216 145 74

216 145 74

216 145 74

Com S 430
The Controller Form

- Scrollbars
- Labels
- Panel 3D
- Exit Button

Com S 430
The RGB Form

Labels
The SUM Class

class Sum : System.Windows.Forms.Form
{
    // Required designer variable.
    private System.ComponentModel.Container components = null;

    private System.Windows.Forms.Label fBlueLabel;
    private System.Windows.Forms.Label fGreenLabel;
    private System.Windows.Forms.Label fRedLabel;

    private Controller fController;

    private bool fDoUpdate;

    private Controller.UpdateColor fUpdateColorHandler;  // event handler
}
The SUM Class Constructor

```csharp
public Sum( Controller aController, int aRed, int aGreen, int aBlue ) : base()
{
    // Required for Windows Form Designer support
    InitializeComponent();

    fController = aController; // register controller
    fUpdateColorHandler =
        new Controller.UpdateColor( ColorChangedHandler );
    // register event handler
    fController.OnColorChanged += fUpdateColorHandler;
    Red = aRed;                  // set Red, no update
    Green = aGreen;              // set Green, no update
    fDoUpdate = true;
    Blue = aBlue;                // set Blue, redraw Form
}
```
InitializeComponent()

private void InitializeComponent()
{
    this.fBlueLabel = new System.Windows.Forms.Label();
    this.fGreenLabel = new System.Windows.Forms.Label();
    this.fRedLabel = new System.Windows.Forms.Label();
    this.SuspendLayout();

    // fBlueLabel
    this.fBlueLabel.Font = new System.Drawing.Font(...);
    this.fBlueLabel.Location = new System.Drawing.Point(0, 84);
    this.fBlueLabel.Name = "fBlueLabel";
    this.fBlueLabel.Size = new System.Drawing.Size(144, 42);
    this.fBlueLabel.TabIndex = 2;
    this.fBlueLabel.Text = "BLUE";
    this.fBlueLabel.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
    this.fBlueLabel.DoubleClick += new System.EventHandler(this.CloneWindow);
    ...
}
InitializeComponent() Cont.

... 
this.ClientSize = new System.Drawing.Size(144, 126);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.fBlueLabel,
    this.fGreenLabel,
    this.fRedLabel});

this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedToolWindow;
this.MaximizeBox = false;
this.MinimizeBox = false;
this.Name = "Sum";
this.Text = "RGB";
this.ResumeLayout(false);
}

Com S 430
**InitializeComponent()**

◆ The method `InitializeComponent()` is a private non-virtual method generated by Visual Studio .NET.

◆ This method creates all of the Form’s controls, set the user-defined properties, and adds all controls to the Form’s Controls property.

◆ The way controls are created and configured is similar to the approach used for JAVA’s AWT objects.
Changing a Color

```csharp
public int Green // Red and Blue similar
{
    get
    {
        return Int32.Parse( fGreenLabel.Text ); // read string
    }

    set
    {
        fGreenLabel.Text = value.ToString();
        UpdateColor(); // redraw Form if necessary
    }
}
```
private void UpdateColor()
{
    if ( fDoUpdate ) // is redraw enabled
    {
        Color lColor = Color.FromArgb ( Red, Green, Blue );
        fRedLabel.BackColor = lColor; // set color of label
        fGreenLabel.BackColor = lColor;
        fBlueLabel.BackColor = lColor;
        Color lComplementColor = Color.FromArgb( 255 - Red,
                                                255 - Green,
                                                255 - Blue );

        fRedLabel.ForeColor = lComplementColor; // set color of text
        fGreenLabel.ForeColor = lComplementColor;
        fBlueLabel.ForeColor = lComplementColor;
    }
}
private void ColorChangedHandler( Color aColor )
{
    fDoUpdate = false;  // disable redraw
    Red = aColor.R;
    Green = aColor.G;
    fDoUpdate = true;   // enable redraw
    Blue = aColor.B;
}

Event Handler
private void CloneWindow(object sender, System.EventArgs e)
{
    (new Sum( fController, Red, Green, Blue )).Show();
}
Overridden OnClosed()

protected override void OnClosed( EventArgs e )
{
    // unregister event handler
    fController.OnColorChanged -= fUpdateColorHandler;
    // call super method
    base.OnClosed( e );
}
public class Controller : System.Windows.Forms.Form
{
    private System.Windows.Forms.Panel panel1;
    private System.Windows.Forms.Label label1;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.Label label3;
    private System.Windows.Forms.VScrollBar vScrollBar1;
    private System.Windows.Forms.VScrollBar vScrollBar2;
    private System.Windows.Forms.VScrollBar vScrollBar3;
    // Required designer variable.
    private System.ComponentModel.Container components = null;

    public delegate void UpdateColor( Color aColor );
    public event UpdateColor OnColorChanged; // multicast delegate
}
The Controller Constructor

```csharp
public Controller()
{
    // Required for Windows Form Designer support
    InitializeComponent();

    // show first view
    (new Sum(this, Red, Green, Blue)).Show();
}
```

Create a view and show it immediately
Changing a Color

// ScrollBar event handler
private void ValueChanged(object sender, System.EventArgs e)
{
    NotifySubscriber();
}

private void NotifySubscriber()
{
    Color lColor = Color.FromArgb(Red, Green, Blue);
    if (OnColorChanged != null) // Is there at least one view
        OnColorChanged(lColor);
    else
    // No views, simply create and show a new one
        (new Sum(this, Red, Green, Blue)).Show();
}
Controller Color Properties

```csharp
public int Red
{
    get { return vScrollBar1.Value; }
}

public int Green
{
    get { return vScrollBar2.Value; }
}

public int Blue
{
    get { return vScrollBar3.Value; }
}
```

Value represents the current slider position.
Controller Exit

// ExitButton Click event
private void ExitController(object sender, System.EventArgs e)
{
    Close(); // closes main Form, hence terminates application
}
Menus

- The class `System.Windows.Forms.Menu` is the base class for all menu-related classes (`MainMenu`, `MenuItem`, and `ContextMenu`).

- This class is abstract and therefore it cannot be instantiated.

- The class `Menu` defines the basic menu-centric behaviors such as providing access to an individual menu item, cloning menus, merging menus (for MDI applications), and so forth.
MainMenu

- The MainMenu control represents the container for the menu structure of a form.

- A menu is composed of MenuItem objects that represent the individual menu commands in the menu structure. Each MenuItem can be a command for your application or a parent menu for other submenu items.

- To bind the MainMenu to the Form that will display it, assign the MainMenu to the Menu property of the Form.
MenuItem

- The MenuItem class provides properties that enable one to configure the appearance and functionality of a menu item.

- For MenuItem objects displayed in a Multiple Document Interface (MDI) application, you can use the MergeMenu method to merge the menus of an MDI parent form with that of its child forms to create a consolidated menu structure.

- The *Popup* event enables one to perform tasks before a menu is displayed. The *Select* event enables one to perform tasks such as providing detailed help for your application's menu items when the user places the mouse cursor over a menu item. The *Click* event enables one to perform tasks associated with the menu item such as opening a file.
Menu Example

mainMenu1 = new System.Windows.Forms.MainMenu();
menuItem1 = new System.Windows.Forms.MenuItem();
menuItem2 = new System.Windows.Forms.MenuItem();
menuItem3 = new System.Windows.Forms.MenuItem();
menuItem4 = new System.Windows.Forms.MenuItem();
mainMenu1.MenuItems.AddRange(new MenuItem[] { menuItem1, menuItem3 });

menuItem1.Index = 0;
menuItem1.MenuItems.AddRange(new MenuItem[] { menuItem2 });
menuItem1.Text = "&File";

menuItem2.Index = 0;
menuItem2.Text = "&Exit";
menuItem2.Click += new EventHandler(menuItem2_Click);

Com S 430
MainMenu Example

menuItem1

menuItem2

menuItem3
private void menuItem2_Click(object sender, System.EventArgs e) {
    Close();
}

Close Form (exit application)
ToolTip

The ToolTip class allows one to provide help to users when they place the mouse cursor over a control.

The ToolTip class is typically used to alert users to the intended use of a control. For example, one could specify ToolTip text for a TextBox control that accepts a name, specifying the format of the name to typed into the control.

In addition to providing help, one can also use the ToolTip class to provide run time status information. For example, one could use the ToolTip class to display connection speed and QOS information when the user moves the mouse cursor over a Control that displays Internet connection status.
ToolTip Example

// create a ToolTip for all controls
fMainToolTip = new System.Windows.Forms.ToolTip(components);

// StatusBar
fMainToolTip.SetToolTip(statusBar1, "Game status.");

// GroupBox
fMainToolTip.SetToolTip(groupBox1, "Card panel.");

// CheckBox
fMainToolTip.SetToolTip(checkBox5, "…");

// PictureBox
fMainToolTip.SetToolTip(pictureBox5, "…");
Main ToolTip Example
The class System.Windows.Forms.Button represents a Windows button control.

A Button may be clicked by using the mouse, ENTER key, or SPACE BAR if the button has focus.

Set the AcceptButton or CancelButton property of a Form to allow users to click a button by pressing the ENTER or ESC keys even if the button does not have focus. This gives the form the behavior of a dialog box.

When a form is shown using the ShowDialog method, one can use the DialogResult property of a button to specify the return value of ShowDialog.
Button Example

```csharp
fDealButton = new Button();

fDealButton.Location = new System.Drawing.Point(200, 176);
fDealButton.Name = "fDealButton";
fDealButton.TabIndex = 5;
fDealButton.Text = "Deal";
fMainToolTip.SetToolTip(fDealButton, "Deal cards");
fDealButton.Click += new System.EventHandler(fDealButton_Click);
```
A Button Event Handler

```csharp
private void fDealButton_Click(object sender, System.EventArgs e) {
    if (fDealButton.Text.Equals("Deal")) {
        // Start a new game...
        fPlayer.NewGame();
        SetCardPictures(fPlayer.Hand);
        fBetComboBox.Enabled = false;
        fDealButton.Text = "Draw";
    } else {
        // draw replacement cards...
        fPlayer.ReplaceCards(BuildHoldString());
        SetCardPictures(fPlayer.Hand);
        fPlayer.ReturnCards();
        // update credit...
        fDealButton.Text = "Deal";
        fBetComboBox.Enabled = true;
    }
}
```
Deal Button Example
ImageList

- ImageList provides methods to manage a collection of Image objects. This class cannot be inherited (i.e. this class is sealed).

- ImageList is typically used by other controls, such as the ListView, TreeView, or ToolBar. One can add bitmaps, icons, or meta files to the ImageList, and the other controls are able to use the images as they require.

- ImageList uses a handle to manage the list of images. The Handle is not created until certain operations, including getting the Images, getting the Handle, and calling Draw are performed on the image list.
ImageList Example

```csharp
ImageList imageListClubs = new ImageList(components);
ImageList imageListSpades = new ImageList(components);
ImageList imageListHearts = new ImageList(components);
ImageList imageListDiamonds = new ImageList(components);

// imageListClubs
imageListClubs.ImageSize = new System.Drawing.Size(71, 96);
```
private void SetCardPictures( IHand aHand ) {
    char[] lDelimiter = { ' ' };
    string[] lCardNames = aHand.Text.Split( lDelimiter );

    for (int i = 0; i < lCardNames.Length; i++) {
        ICard aCard = aHand[lCardNames[i]];
        switch ( aCard.Suit ) {
            case 0: fPictures[i].Image =
                fImageListClubs.Images[aCard.Number - 2];
                break;
            ...
            case 3: fPictures[i].Image =
                fImageListSpades.Images[aCard.Number - 2];
                break; }
    } } 
Loading Images
Panel

- A Panel is a control that contains other controls.
- You can use a Panel to group collections of controls such as a group of RadioButton controls.
- If the Panel control’s Enabled property is set to false, the controls contained within the Panel will also be disabled.
- The Panel control is displayed by default without any borders. You can provide a standard or three-dimensional border using the BorderStyle property to distinguish the area of the panel from other areas on the form.
Panel Example

// WinTicTacToe Board
panel1 = new Panel();
panel1.BorderStyle = BorderStyle.Fixed3D;
panel1.Controls.AddRange(new Control[] {panel7, panel5,
                                          panel4, panel2,
                                          label_3_3, label_3_2,
                                          label_3_1, label_2_3,
                                          label_2_2, label_2_1,
                                          label_1_3, label_1_2,
                                          label_1_1});
panel1.Location = new System.Drawing.Point(10, 10);
panel1.Name = "panel1";
panel1.Size = new System.Drawing.Size(194, 194);
panel1.TabIndex = 1;
WinTicTacToe Board
GroupBox

- The.GroupBox displays a frame around a group of controls with or without a caption.

- Use a GroupBox to logically group a collection of controls on a form. The group box is a container control that can be used to define groups of controls.

- The typical use for a group box is to contain a logical group of RadioButton controls. If you have two group boxes, which each contain several radio buttons, both groups of buttons would be mutually exclusive and set two individual option values.

- Only controls contained within the GroupBox control can be selected or receive focus. The entire GroupBox itself cannot be selected or receive focus.
GroupBox Example

groupBox2 = new GroupBox();
groupBox2.Controls.AddRange(new Control[] { fInfoLabel, fBetComboBox, fCreditsLabel, label2, label1 });
groupBox2.Location = new System.Drawing.Point(16, 16);
groupBox2.Name = "groupBox2";
groupBox2.Size = new System.Drawing.Size(476, 88);
groupBox2.TabIndex = 2;
groupBox2.TabStop = false; // cannot receive focus
groupBox2.Text = "Credits";
fMainToolTip.SetToolTip(groupBox2, "Credit panel.");
WinRemPoker

Click Deal to start

Credits: 0

Bet: 1

groupBox2

fCreditsLabel

fInfoLabel

fBetComboBox
The TextBox control allows the user to enter text in an application.

Typically, a TextBox control is used to display, or accept as input, a single line of text. You can use the Multiline and ScrollBars properties to enable multiple lines of text to be displayed or entered.

You can limit the amount of text entered into a TextBox control by setting the MaxLength property to a specific number of characters. TextBox controls can also be used to accept passwords and other sensitive information. You can use the PasswordChar property to mask characters entered in a single-line version of the control.
TextBox Example

```csharp
fMessages = new TextBox();
fMessages.Location = new System.Drawing.Point(16, 24);
fMessages.Multiline = true;
fMessages.Name = "fMessages";
fMessages.ReadOnly = true;
fMessages.Size = new System.Drawing.Size(576, 136);
fMessages.TabIndex = 0;
fMessages.TabStop = false;
fMessages.Text = "The other\'s messages";
```
If a TextBox’s ReadOnly property is set to true, then the control is displayed using the standard color for controls.
StatusBar

- Typically a StatusBar control consists of a set of StatusBarPanel objects, each of which displays text and/or an icon.

- A StatusBar control typically displays information about an object being viewed on a Form, the object's components, or contextual information that relates to that object's operation within your application.

- The default StatusBar has no panels.
StatusBar Example

```csharp
StatusBar fStatusBar = new StatusBar();
StatusBarPanel fStatusBarPanelName = new StatusBarPanel();
StatusBarPanel fStatusBarPanelOnline = new StatusBarPanel();
fStatusBar.Location = new System.Drawing.Point(0, 259);
fStatusBar.Name = "fStatusBar";
fStatusBar.ShowPanels = true;
fStatusBar.Size = new System.Drawing.Size(632, 22);
fStatusBar.TabIndex = 2;
fStatusBar.Panels.AddRange(new StatusBarPanel[] { fStatusBarPanelName, fStatusBarPanelOnline });

fStatusBarPanelName.Text = "Your name";
fStatusBarPanelName.Width = 300;
fStatusBarPanelOnline.Alignment = HorizontalAlignment.Center;
fStatusBarPanelOnline.Text = "Offline";
fStatusBarPanelOnline.Width = 300;
```
ChatClient Statusbar

In order to display information in a status bar one needs to create status bar panels. Furthermore, the StatusBar’s ShowPanels property must be set to true.