COMP 585 Noteset #10
More C#/WinForms

Layout

- Layout Event
- LayoutEngine Class (System.Windows.Forms.Layout namespace)
- Panel Control
- SplitterPanel Control
- FlowLayoutPanel Control
- TableLayoutPanel Control
- TabControl and TabPage for Tabbed Pages

Some Additional WinForm GUI elements

- DateTimePicker: composite control for constructing dates and times
- NumericUpDown: called JSpinner in Swing
- MenuItem and MainMenu: called JMenuBar, JMenu, and JMenuItem in Swing
- TreeView and TreeNode: called JTree and DefaultMutableTreeNode in Swing
- TrackBar: called JSlider in Swing

Class SplitContainer (like JSplitPane in Swing)

```
SplitContainer split = new SplitContainer();
split.Parent = this;
split.Dock = DockStyle.Fill;

TreeView tree = new TreeView();
tree.Parent = split.Panel1;
tree.Dock = DockStyle.Fill;
tree.Nodes.Add("tree");

ListView list = new ListView();
list.Parent = split.Panel2;
list.Dock = DockStyle.Fill;
list.Items.Add("list");
```

Class FlowLayoutPanel (like JPanel with FlowLayout in Swing)

```
FlowLayoutPanel flow = new FlowLayoutPanel();
flow.Parent = this;
flow.Dock = DockStyle.Fill;
flow.Text = "Flow Panel";
flow.Click += ClickHandler;

Random rand = new Random(DateTime.Now.Millisecond);

for (int i = 0; i < 20; i++)
{
    Button btn = new Button();
    btn.Parent = flow;
    btn.Text = "Button " + (i + 1);
    btn.Click += ClickHandler;

    // Set a random size (but not too random)
    Size sz = btn.PreferredSize;
    sz.Width = (int)(sz.Width * (1 + 2 * rand.NextDouble()));
    sz.Height = (int)(sz.Height * (1 + 2 * rand.NextDouble()));
    btn.Size = sz;
}
```

Scrolling

There is no separate control for scrolling. Forms and some other controls have the ability to scroll automatically if their client area needs scrolling.

```
Form f = new Form();
f.AutoScroll = true;
```

Just as in Java Swing, automatic scrolling only works if the form can calculate the size of the information to display. If a series of controls have been assigned to the form, then the size is known, and scrolling can be turned on with AutoScroll. But if the content of the form is derived from graphics that are painted or drawn, the form does not automatically know how much space is needed to display the graphic or image.

For scrolling images, one solution is to introduce a PictureBox control to display the image. Since the control is added to the form, the form can now correctly autoscroll to display the image.

```
public class ImageForm : Form {
      public Image img;
      public PictureBox pb;

      public ImageForm() : base() {
            img = Image.FromFile("img02.jpg");
            pb = new PictureBox();
            pb.Image = img;
            pb.ClientSize = new Size(img.Width,img.Height);
            pb.Parent = this;
            this.AutoScroll = true;
      }

      public static void Main() {
            ImageForm form = new ImageForm();
            Application.EnableVisualStyles();
            Application.Run(form);
      }
}
```

To directly scroll the image to display an image without a PictureBox, individual scroll bar controls must be introduced. Here's a skeleton code:

```
class ImageTest : Form {
      public Image img;
      public HScrollBar hs;
      public VScrollBar vs;

      public ImageForm() : base() {
            img = Image.FromFile("img02.jpg");

            hs = new HScrollBar();
            hs.Parent = this;
            hs.Minimum = 0;
            AdjustHS();
            hs.ValueChanged += HSChange;
            hs.Dock = DockStyle.Bottom;

            vs = new VScrollBar();
            vs.Parent = this;
            vs.Minimum = 0;
            AdjustVS();
            vs.ValueChanged += VSChange;
            vs.Dock = DockStyle.Right;

            this.DoubleBuffered = true;
      }
```

```
        public void AdjustHS() {
              hs.Maximum = img.Width - this.ClientSize.Width;
              if (hs.Maximum < 0) hs.Maximum = 0;
              if (hs.Value > hs.Maximum) hs.Value = hs.Maximum;
        }

        public void AdjustVS() {
              vs.Maximum = img.Height - this.ClientSize.Height;
              if (vs.Maximum < 0) vs.Maximum = 0;
              if (vs.Value > vs.Maximum) vs.Value = vs.Maximum;
        }
  }
```

There are 4 event handlers required.
For the form:
- OnPaint:
  - Redraw the image based on the client size and scrollbar positions
- OnResize:
  - Adjust the scrollbars, and call Invalidate() to trigger a repaint

For the Horizontal ScrollBar:
- OnValueChanged:
  - Call Invalidate to trigger a repaint

For the Vertical Scroll Bar:
- OnValueChanged:
  - Call Invalidate to trigger a repaint

When the form is resized, the maximum extent of each scrollbar must be updated based on a consideration of the new size of the form and the constant size of the image. Inside OnPaint(), most of the work is done by the DrawImage of class Graphics. There are a lot of overloaded versions of this method, some of which introduce scaling factors. Here's one way to get the right effect:

```
  protected override void OnPaint(PaintEventArgs args) {
        base.OnPaint(args);
        Graphics g = args.Graphics;
        g.DrawImage(
              img,
              new Rectangle(0,0,this.Width,this.Height),
              new Rectangle(hs.Value,vs.Value,this.Width,this.Height),
              GraphicsUnit.Pixel
        );
  }
```

The 1st rectangle defines where on the form to draw the image, in form coordinates. The 2nd rectangle defines what part of the image to draw, in image coordinates. The code shows that from the perspective of the image, the upper left corner from which to start drawing is indicated by the current values of the scroll bars. The width and height for the draw are the width and height of the form, not the image. Basically, the 2nd rectangle defining the image is mapped onto the 1st rectangle defining the client area of the form.

Imagine what happens when the horizontal scrollbar is advanced one click. This causes its value to increase by one, and a repaint to be triggered. The repaint redraws the image, this time one pixel further to the right within the image.

Class TreeView and TreeNode

The TreeView control in Windows Forms is analogous to the JTree component in Java Swing. It displays a collection of data arranged into a tree, where each node is an instance of class TreeNode (which is analogous to class DefaultMutableTreeNode in Java Swing).

A TreeNode control is created, then nodes are added to its collection of nodes via the property named "Nodes".

```
TreeView tree = new TreeView();
TreeNode n = new TreeNode("node 1");
tree.Add(n);              // WRONG
tree.Nodes.Add(n);        // CORRECT
```

Nodes can also be added to the "Nodes" collection of other nodes:

```
TreeNode a = new TreeNode("a");
TreeNode b = new TreeNode("b");
a.Nodes.Add(b);
```

The meaning of root node for a Windows Forms "TreeView" is slightly different than that for Java Swing "JTree". All the nodes that are added directly to the TreeView control are called "root nodes". A TreeView control can have multiple root nodes at the top of the tree without a common parent node. You can think of all nodes added directly to the TreeView as being child nodes of an implicit root node, but you cannot access or modify this imaginary or virtual root. For consistency with other applications, the programmer can choose to have only one root node added directly to the control, and all other nodes are child nodes of this one root node, but the user would have to enforce such a convention.

Here's a more complete example:

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

class TreeTest : Form {
      public TreeTest() {

            // Build array of 4 customers:  0 thru 3

            ArrayList customers = new ArrayList();
            for (int i=0; i<4; i++)
                  customers.Add("customer " + i);


            // For each customer, build an array of orders
            // 0:3   1:6   2:0   3:2

            ArrayList order0 = new ArrayList();
            for (int i=0; i<3; i++)
                  order0.Add("order " + i);
            ArrayList order1 = new ArrayList();
            for (int i=0; i<6; i++)
                  order1.Add("order " + i);
            ArrayList order2 = new ArrayList();
            ArrayList order3 = new ArrayList();
            for (int i=0; i<2; i++)
                  order3.Add("order " + i);

            // Create TreeView Control
            TreeView tree = new TreeView();
```

```
                // Create a Node for each customer and add to the TreeView
                // For each customer, create a Node for each order, and
                // add the order to its customer's Node

                for (int i=0; i<customers.Count; i++) {
                      TreeNode n = new TreeNode((string)customers[i]);
                      tree.Nodes.Add(n);
                      if (i==0) for (int j=0; j<order0.Count; j++)
                            n.Nodes.Add(new TreeNode((string)order0[j]));
                      else if (i==1) for (int j=0; j<order1.Count; j++)
                            n.Nodes.Add(new TreeNode((string)order1[j]));
                      else if (i==2) for (int j=0; j<order2.Count; j++)
                            n.Nodes.Add(new TreeNode((string)order2[j]));
                      else if (i==3) for (int j=0; j<order3.Count; j++)
                            n.Nodes.Add(new TreeNode((string)order3[j]));
                }

                // attach the TreeView to the Form
                tree.Parent = this;
          }
    }

    class Driver {
          public static void Main() {
                TreeTest form = new TreeTest();
                Application.Run(form);
          }
    }
```

In this example, the data stored at each node is of type "string", but it can be of any type. Note that the TreeView has automatic scroll capability which is triggered when a node is expanded. Basically it comes with its own scroll pane (more about that later).

Clicking on a TreeNode

Clicking a TreeNode triggers a NodeMouseClick event on behalf of the TreeView. To respond, provide a handler:

```
TreeView tree = …;
tree.NodeMouseClick += delegate(object obj, TreeNodeMouseClickEventArgs args) {
      TreeNode node = args.Node;
      Console.WriteLine("Clicked on node " + node.Text);
};
```

The event argument provides a property that indicates the clicked node. There is no direct representation of the path from the root to the selected node, but this info can be recovered from the Parent property of each TreeNode:

```
      TreeNode node = args.Node;
      while (node != null) {
            … // action for current node here
            node = node.Parent; // move up the tree one level toward the root
      }
```

Menus
A form, like a Java JFrame, can display a menu bar. Just like in Java, menus are composed of menu items. The menu bar displayed at the top of the form just below the title bar is the form's main menu (as opposed to a popup menu, called a shortcut or context menu here). Windows Forms terminology for describing the menu system is a little different from the nomenclature used in the Java Swing tutorial.

- The menu bar is referred to as "the main menu" (class MainMenu)
- The title of each menu is referred to as a "top-level" menu item (MenuItem)
- The menu itself which opens when its title is clicked is referred to as a "popup menu" or "drop-down menu", or more recently "submenu" or "child menu".
- A popup menu is referred to as a "context menu" (class ContextMenu).

A MainMenu object is usually associated with the application's main form, while a ContextMenu is usually associated with a particular control on the form.

Classes MainMenu, MenuItem, and ContextMenu derive from abstract class Menu, which itself **does not** derive from class Control.

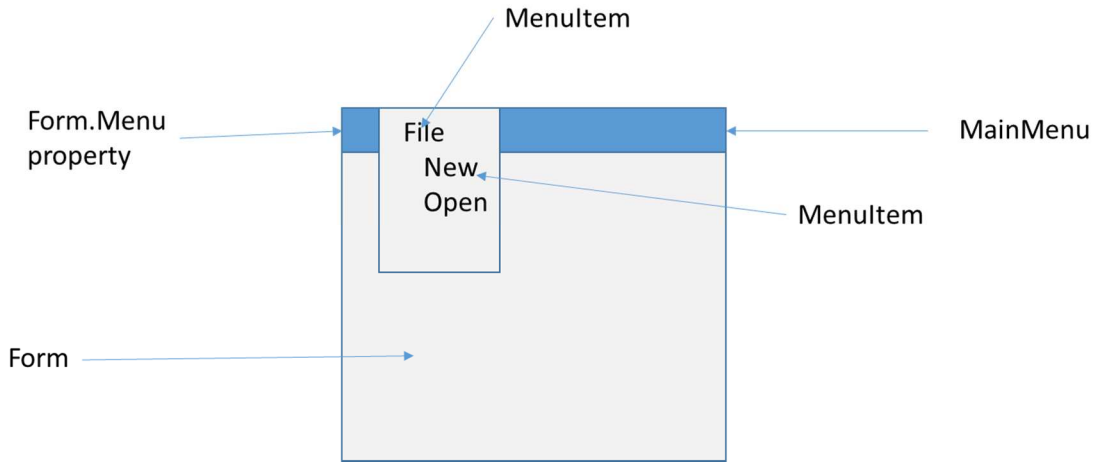
To build a menu use one of the constructors:
- MainMenu()
- MainMenu(MenuItem[] items)

To attach the main menu to the form, use the form's **Menu** property.

To build a context menu, constructors are similar.
To attach a context menu to a control, use the control's **ContextMenu** property. Alternatively, you can handle the context menu in the OnMouseUp event handler, and respond based on current mouse position.

Menu Item Constructors
- MenuItem()
- MenuItem(string txt)
- MenuItem(string txt, EventHandler handler)
- MenuItem(string txt, EventHandler handler, Shortcut sc)
- MenuItem(string txt, MenuItem[] items)

The last one would be normally used only for the top-level menu item.



Here's a simple example that creates a simple main menu (ignoring event handling for now):

```
class MenuForm : Form {
      public MenuForm() {

            // Build the File Menu
            MenuItem[] miFileItems = new MenuItem[3];
            miFileItems[0]  = new MenuItem("New");
            miFileItems[1]  = new MenuItem("Open");
            miFileItems[2]  = new MenuItem("Quit");
            MenuItem miFile = new MenuItem("&File", miFileItems);

            // Build the Edit Menu (see alternative #2 below)
            MenuItem[] miEditItems = new MenuItem[3];
            miEditItems[0] = new MenuItem("Cut");
            miEditItems[1] = new MenuItem("Copy");
            miEditItems[2] = new MenuItem("Paste");
            MenuItem miEdit = new MenuItem("&Edit", miEditItems);

            // Build the Main Menu
            MenuItem[] mmItems = new MenuItem[2];
            mmItems[0] = miFile;
            mmItems[1] = miEdit;

            // Attach the Main Menu to the Form (see alternative #1 below)
            MainMenu mmenu = new MainMenu(mmItems);
            this.Menu = mmenu;
      }

      public static void Main() {
            Application.EnableVisualStyles();
            Application.Run(new MenuForm());
      }
}
```

Alternative #1: You can add top-level menu items to the main menu after it has been created by using its **MenuItems** property.

```
            MainMenu mmenu = new MainMenu();
            mmenu.MenuItems.Add(miFile);
            mmenu.MenuItems.Add(miEdit);
```

Alternative #2: Same goes for a top-level menu item.

```
            MenuItem miEdit = new MenuItem("&Edit");
            miEdit.MenuItems.Add(miEditItems[0]);
            miEdit.MenuItems.Add(miEditItems[1]);
            miEdit.MenuItems.Add(miEditItems[2]);
```

Menu Strip Control
An updated approach to building menus in WinForms is to use the **MenuStrip** control in place of the MainMenu control, and use **ToolStripMenuItem** controls in place of MenuItem controls.

```
public class MenuDemo : Form {

      public MenuDemo() : base() {
            MenuStrip ms = new MenuStrip();

            // File Menu
            ToolStripMenuItem miFile = new ToolStripMenuItem();
            miFile.Text = "File";

            // File Menu items
            ToolStripMenuItem miNew = new ToolStripMenuItem();
            miFile.DropDownItems.Add(miNew);
            miNew.Text = "New";
            ToolStripMenuItem miOpen = new ToolStripMenuItem();
            miOpen.Text = "Open";
            miFile.DropDownItems.Add(miOpen);

            // attach menus to menu strip
            ms.Items.Add(miFile);

            // attach menu strip to form
            ms.Dock = DockStyle.Top;
            this.Controls.Add(ms);        // add last for z-order issues
      }
      …
}
```

Activating Menu Items
To respond to a user click on a menu item, attach or register a delegate with the Click event for the menu item. Note that in this case, it will be more common to register a delegate than to subclass the MenuItem class in order to override the "built-in" OnClick() method.

The delegate for the Click event is the generic one:

```
delegate void EventHandler(object obj, EventArgs args);
```

Just add the following code to the example shown above. First, after the menu item of interest has been defined, register a delegate for the Click event. In this case, let's add a handler for the "Cut" menu item:

```
miEditItems[0].Click += CutHandler;
```

Second, define the actual handler somewhere in the body of the class:

```
public void CutHandler(object obj, EventArgs args) {
      MessageBox.Show("You chose the Cut Menu Item.");
}
```

Since this is an external handler method (not an override of the OnClick() method), the handler has two parameters, the first of type object, and the second of type EventArgs.

Menu Separators
Use a "dash" as the text for a menu item to create a menu separator.

```
MenuItem separator = new MenuItem("-");
```

Menu Shortcuts
A menu item may be selected from the keyboard by using a keystroke consisting of the alt key plus some letter that opens the menu (if it is a top-level menu item) or selects a menu item (all other menu items). This was called a mnemonic in Java Swing.

The letter to be used is preceded by an ampersand "&" in the text name of the menu item. That letter will be underlined when the ALT key is pressed when the application is running (note that this is different from Java Swing).

In addition, a menu item can be given a **shortcut** (called an accelerator in Java Swing). The **Shortcut** enumeration explicitly defines all possible menu item shortcuts. You cannot define arbitrary keystrokes to use as shortcuts as you can in Java. The shortcut for a menu item can be specified via the constructor, or it can be assigned to the Shortcut property for the menu item.

```
MenuItem miCut = new MenuItem("Cu&t");
miCut.Shortcut = Shortcut.CtrlX;
```

Customized Menu Items
An interesting feature of Windows Forms menu items is the **bool OwnerDraw** property. The default value is false; if set to true, the programmer provides a delegate event handler for the **DrawItem** event, and uses standard graphics methods to draw the menu item in whatever customized way is desired.

Dialogs
Goal of GUI is to make features of application available to the user of the application. This is not a challenge if there are only a handful of features, but if there are hundreds, the access mechanism must keep the features well-organized and be intuitive to navigate.

Usually, a menu system meets most of these goals. But if a particular feature require the gathering of a lot of specialized info, or if the number of options to be set is large, a dialog may be required. In response to selecting some menu items, a dialog is opened to allow info to be gathered or options to be set, before the actual feature is executed.

A dialog is like a second form or window that opens when needed. Dialogs come in two varieties:

- **Modal** (most common): once open, user cannot navigate away to another window in the same app until the dialog is closed or dismissed.
- **Modeless**: user can leave the dialog open, navigate to another window, and return later to eventually close it.

Information in a dialog is relatively free form, but it's a widely accepted convention for it to provide "OK" and "Cancel" buttons near the bottom. Unlike Java Swing, C#/WinForms provides no special dialog class, it is constructed as a derived class from class Form, just like the main window of an application. What makes a dialog a dialog is how it is used, not its base class. The following example is taken from an earlier WinForm textbook by Charles Petzold. The constructor for the dialog has the following settings for Form properties:

```
FormBorderStyle = FormBorderStyle.FixedDialog;
ControlBox      = false;
MaximizeBox     = false;
MinimizeBox     = false;
ShowInTaskbar   = false;
```

These settings enforce additional conventions that differentiate a Form acting as a dialog from a Form acting as an application main window. These settings prevent the dialog from being resized, minimized, maximized, or displayed on the task bar at the bottom of the desktop. It also prevents the user from closing the dialog by clicking on the "X" in the upper left corner (the close box). To close a dialog, you normally click the "OK" or "Cancel" button.

Making a Dialog Visible
C#/WinForms provides some interesting support for creating a dialog out of forms, buttons, and events.

Since a dialog is a type of Form, you must command the dialog to "show" itself after it has been instantiated in order for it to be visible. There is a special method of class Form for this purpose named **ShowDialog().** This is used for modal dialogs.

This method does not return until the dialog has been closed. The most common way to trigger the display of a dialog is to select a menu item. So the call to ShowDialog() will usually appear inside the Click event handler for some menu item. There is even a common convention when naming a menu item to add the ellipsis sequence "…" to the text of the menu to indicate that a dialog will be opened when this menu item is selected. This is a convention not a requirement.

Dismissing a Dialog
As mentioned above, a dialog normally has "OK" and "Cancel" buttons that are used to close the dialog when the user is finished. These buttons are given handlers that dismiss the dialog by assigning a value to a special form property named **DialogResult.** Even after a dialog has been closed, it still exists as an object within the program and can be shown again at a later time. Importantly, after it has been closed, other code can check its **DialogResult** property to determine how it was closed.

The possible values that the **DialogResult** property can receive are defined by the **DialogResult enumeration**: **None, OK, Cancel, Abort, Retry, Ignore, Yes, No**. The names in this enumeration correspond to text commonly used on dialog dismiss buttons. You should make sure that clicking a button with a specific text label causes the corresponding dialog result value to be assigned to the **DialogResult** property.

Some Shortcuts
The ShowDialog() method returns the value assigned to the DialogResult property. This can simplify dialog handling by branching based on this return value.

Class Button also defines a DialogResult property, just like class Form. Many dialogs only use the "OK" or "Cancel" buttons to close the dialog. When creating the "OK" and "Cancel" buttons, you can define the DialogResult property for the button, and avoid defining Click event handlers for the buttons. When the dialog is closed by clicking one of the buttons, the DialogResult property is copied to the same property of the parent dialog. Back in the menu Click handler, the ShowDialog() method gets a return value that reflects which button was clicked in order to close the dialog, as before.

Screen Location
Whenever any form is first created and made visible, its start position is controlled by its **StartPosition** property.

- **StartPosition** Property: type **FormStartPosition**
- **FormStartPosition** Enumeration: Manual, CenterScreen, WindowsDefaultLocation, WindowsDefaultBounds, CenterParent

If a main form opens a dialog box (subordinate form), the recommended approach is to set the FormStartPosition to Manual. This allows the dialog box to position itself relative to the main form with the most flexibility. The easiest way for the dialog box form to know the location of the main application form is to use the ActiveForm property, which is a static property of class Form. Add the following to the standard dialog property settings given above:

```
this.StartPosition = FormStartPosition.Manual;
this.Location      = Form.ActiveForm.Location + …; // plus some small offset
```

The location and size of the dialog box should not be identical to the main application form, or it will hide it, which could be disorienting to the user.

Moving Info from Dialog Box to Main Application Form
The purpose of a dialog box is to collect info from the user, who will interact with the dialog – enter text, move sliders, select radios or check boxes – to input information. The dialog collect the info on behalf of the main application form.

After the dialog box has been dismissed, the main application form can now obtain info from the controls attached to the dialog box. Rather than forcing the application code to search through controls to find the values of interest, the programmer should provide definitions of appropriate properties for the dialog box class, which can then be easily queried by the application code.

The Apply Button
Classical dialogs used "OK" and "CANCEL" buttons to dismiss with or without enacting changes. Such dialogs are naturally modal. The relationship between the main form and the dialog box was simple. Code within the main form could examine the state or properties of the dialog box after it was dismissed to obtain the results of the user interaction.

Introduction of modeless dialogs with an "APPLY" button represented an innovation to solve GUI problems for functions that required a dialog but for which the dialog should remain on the desktop and not control the focus. For example, think of implementing the find/replace function in a text editor with a modal dialog. Modeless dialogs are more convenient for the application's end user but they complicate the app's logic by requiring communication between the main form and the dialog before the dialog is dismissed. The dialog box code should not be written to call a method belonging to the application form class, because it would require the application code to define a callback method for the dialog to use. It's better for the dialog to present a set of properties and events that can be used by the application form in whatever way seems appropriate. This solution raises another complication: the only way for the application form to know when the "APPLY" button has been clicked is for the dialog box to fire and event which the application form will handle.

The goal here is to make a dialog box modular and reusable. The strategy is as follows:

- The dialog box provides an "Apply" button.
- The dialog box also provides a **bool ShowApply** property that allows the user of the dialog to turn this feature on or off.
- The dialog box also defines its own unique event type, which will be fired when the Apply button is clicked. The code to define the event is:

```
public event EventHandler Apply;
```

  which defines an event named **Apply**, which is based on the **EventHandler** delegate. The code to fire the event is:

```
if (Apply != null)
      Apply(this, new EventArgs());
```

- The main application form then registers an event handler with the Apply event for that dialog box.

When the user selects the "Color…" menu item on the main form, the handler for the menu item is called. The handler does the following:

- Create the dialog with the apply button enabled
- Register an Apply event handler
- Call the ShowDialog() method for the dialog box
- Wait until it returns and check the return value.
- If the return value indicates the "OK" button was called, read properties from the dialog box and assign those values to the correct variables within the application form.

In addition to this normal processing for a dialog box, clicking the Apply button triggers an Apply event, which causes the Apply handler to be called. The Apply event handler executes the same code that is executed if the OK button is clicked. The two cases are handled by different event monitoring paths.

Modeless Dialogs

Having better communication between form and dialog box with an Apply button is an improvement in flexibility, but some situations call for a modeless dialog, a dialog that does not control the focus until it is dismissed. Applications commonly allow the user to leave some dialog boxes visible while switching back and forth between the main form and the dialog. Example: the find/replace dialog in a word processor.

There are a few steps that the main form must perform for a modeless dialog that didn't apply in the case of a modal dialog. For example, after the main form creates the dialog, it sets its **Owner** property to itself ("this"). This ensures that the dialog always appears in front of the main form, and that if the main form is minimized, the dialog disappears along with the main form.

To make the dialog visible, call the **Show()** method rather than the **ShowDialog()** method.

To dismiss the dialog, call the Hide() method. If you plan to Show() the same dialog later, don't call the Close() method, since this disposes of the object and its resources.

Predefined Dialogs
A simple custom modal or modeless dialog can be defined as a derived class of the Form base class. Starting with Windows 3.1, a standard set of predefined dialogs was added to the Windows API and these are still available in the .NET class library, System.Windows.Forms namespace:

- ColorDialog
- OpenFileDialog
- SaveFileDialog
- FontDialog
- PageSetupDialog
- PrintDialog

These are all derived classes from the abstract base class CommonDialog. Check the API for properties and methods. For example, to use the OpenFileDialog, instantiate it and set the desired property values:

```
OpenFileDialog opendlg = new OpenFileDialog();
```

In the delegate that shows the dialog, treat it as a regular modal dialog:

```
DialogResult result = opendlg.ShowDialog();
```

After the dialog is dismissed, examine the FileName property to get the name of the selected file as a string:

```
String filename = opendlg.FileName;
```

To access the file, use the classes from the System.IO namespace

- **StreamReader** to read from the file
- **StreamWriter** to write to the file

There are many other possible classes in the namespace for specific IO problems.

The [STAThread] Attribute
Software development for Windows contains a few hooks that are required to maintain compatibility with several coding models that have been introduced over the years. One of those models is called Component Object Model or COM. It is an interface at the binary level that allows objects from separate compilations to communicate at runtime. COM objects can use one of two threading models called Single Threaded Apartment (STA) or Multi Threaded Apartment (MTA). This is mostly just an issue for COM programmers.

However, the issue comes up for WinForms programs, because some utility software follows the COM model. If you mix COM objects with WinForm code, WinForms is not compatible with MTA, and the Main() method must be marked with the [STAThread] attribute.

```
[STAThread]
public static void Main() {
        …
}
```

Technically, Microsoft recommends that the entry point for any WinForms app include this attribute. The program may work without it, but when COM objects (like predefined dialogs) are included in a WinForms app, the [STAThread] attribute becomes required, the app will not function correctly without it.

SDI vs. MDI Applications
A GUI which allows only a single document window to be open at a time is called a Single Document Interface (SDI). A GUI which allows multiple document windows to be open at a time, all under the control of a single parent window is called a Multiple Document Interface (MDI). For the MS Office apps, Microsoft has gone back and forth over the years from SDI to MDI and back to SDI. It's an open question among developers which is preferred.

A simple app like Notepad has always been implemented with an SDI-based GUI. By launching multiple instances of the Notepad app, the user can have multiple Notepad document windows open on the desktop at the same time, but this is not what is meant by MDI. A more complex app like Word has been implemented with both SDI and MDI-based GUIs.

In Java, an MDI app can be built using the classes JDesktopPane and JInternalFrame.

Windows Forms allows an MDI GUI to be built by setting a few properties.

- The parent Form container sets the IsMdiContainer property to true.
- The child Forms set their MdiParent property to the parent form.

It's a common convention in MDI GUIs to provide a list of all open windows. This feature can be added by setting one more property – MdiListItem – with either the MainMenu or MenuStrip control, depending on which one your app is using.

Text Editing, Lists, Combo Boxes, Spin

Abstract Class TextBoxBase
Properties
- string Text
- int MaxLength
- int TextLength
- string[] Lines

Derived Classes
- Class TextBox
- Class RichTextBox

Events (from Class Control)
- TextChanged (based on EventHandler delegate)

Properties (from class TextBoxBase)
- int SelectionStart
- int SelectionLength
- string SelectedText
- bool HideSelection

Methods (from class TextBoxBase)
- void Select(int iStart, int iLength)
- void SelectAll()
- void Clear()

The Clipboard
- Default Undo/Redo (single-level only, no arbitrary "stack" of undoable edits)

Properties (from Class TextBox)
- bool Multiline ← diff between a text box and a text pane
- bool WordWrap
- ScrollBars ScrollBars

Scroll Bars
- ScrollBars Enumeration: None, Horizontal, Vertical, Both
- RichTextBoxScrollBars Enumeration: adds ForcedHorizontal, ForcedVertical, ForcedBoth

Properties (from Class TextBox)
- HorizontalAlignment TextAlign
- HorizontalAlignment Enumeration: Left, Right, Center

Cloning Notepad
Here's an example from a previous text by Petzold on WinForms that presents several versions of a text editor that mimics the capabilities of MS Notepad.

- NotepadCloneNoMenu: base class, simple TextBox
- NotepadCloneWithRegistry: uses registry to remember user preferences
- NotepadCloneWithFile: adds File I/O

NotepadCloneNoMenu

- Derived from class **Form**
- One Instance Variable Control of Type **TextBox**
- Set appropriate properties of the TextBox (Dock, Multiline, AcceptsTab, ScrollBars, WordWrap)

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

public class Editor : Form {

      TextBox editarea;

      public Editor() : base() {
            editarea = new TextBox();
            editarea.Dock       = DockStyle.Fill;
            editarea.Multiline  = true;
            editarea.AcceptsTab = true;
            editarea.ScrollBars = ScrollBars.Both;
            editarea.WordWrap   = false;
            this.Controls.Add(editarea);
      }

      public static void Main(string[] args) {
            Editor editor = new Editor();
            Application.EnableVisualStyles();
            Application.Run(editor);
      }
}
```

Files and Streams in .NET

- Defined in namespace System.IO
- Applicable for both Windows Forms and WPF
- This section is off topic for GUIs, so is covered only in bullet form here.

Class FileStream

- Inherits from abstract class **Stream**
- The most basic class for file open, read, write, close.

Some Useful Enumerations

- FileMode Enumeration
- FileAccess Enumeration
- FileShare Enumeration

Properties

- bool CanRead
- bool CanWrite
- bool CanSeek
- bool Length
- bool Position

Stream Methods

- int ReadByte()
- int Read(byte[] buff, int offset, int count)
- void WriteByte(byte value)
- void Write(byte[] buff, int offset, int count)
- long Seek(long offset SeekOrigin origin)
- void SetLength(long length)
- void Flush()
- void Close()

Problem With FileStream

- Limited to reading/writing arrays of bytes.
- Use StreamReader, StreamWriter for text, which supports operations on strings
- Use BinaryReader, BinaryWriter for binary data

Encoding

- Class in System.Text with several predefined static values:
    - Default, Unicode, BigEndianUnicode, UTF8, UTF7, ASCII
- Can also construct an encoding if the predefined ones are not sufficient.

UTF-8 Encoding

- Defined by RFC 2279 at http://www.ietf.org
- Represents Unicode characters without wasting space on zero-valued bytes.
- Each Unicode character is translated to between 1 and 6 non-zero bytes
- Unicode characters in the ASCII range map directly to ASCII.
- UTF = UCS Transformation Format
- UCS = Universal Character Set = ISO 10646

Reading and Writing Text

- StreamWriter for Writing Text
- Inherits from abstract class **TextWriter**

StreamWriter Properties

- Stream BaseStream
- Encoding Encoding
- bool AutoFlush
- string NewLine default = "\r\n", can be changed to "\n"

Class TextWriter Methods

- void Write()
- void WriteLine(…)
- void Flush()
- void Close()

Class StreamReader

- For reading text files

StreamReader Properties

- Stream BaseStream
- Encoding CurrentEncoding

StreamReader Methods

- int Peek()
- int Read()
- int Read(char[] buff, int offset, int count)
- string ReadLine() <== probably the most useful
- string ReadToEnd()
- void Close()

Class Environment

Methods for Info About File System

- string[] GetLogicalDrives()
- string GetFolderPath(Environment.SpecialFolder f)

Static Properties

- string SystemDirectory
- string CurrentDirectory

File and Path Name Parsing

Class Path
Contains static methods and properties to help with file and path name parsing.

Methods
bool isPathRooted(string s)
bool HasExtension(string s)
string GetFileName(string s)
string GetFileNameWithoutExtension(string s)
string GetExtension(string s)
string GetDirectoryName(string s)
string GetFullPath(string s)
string GetPathRoot(string s)

string Combine(string s, string t)
string(ChangeExtension(string s, string ext)

string GetTempPath()
string GetTempFileName()

Fields
char PathSeparator ;
char VolumeSeparatorChar :
char DirectorySeparatorChar \
char AltDirectorySeparatorChar /
char[] InvalidPathChars " < > |

Working With Directories

Directory and DirectoryInfo classes are very parallel

- Directory: **static** methods with extra required string parameter for path
- DirectoryInfo: **non-static** methods with no extra parameter

Two similar ways to get the same information
Both are **sealed** (can't be subclassed) and derive from abstract class **FileSystemInfo**

Static Methods of Class Directory
string[] GetLogicalDrives()
string GetCurrentDirectory()
void SetCurrentDirectory(string s)

Constructor for DirectoryInfo
DirectoryInfo(string s)
Creates a DirectoryInfo object that is associated with a specific physical directory.

DirectoryInfo Properties
bool Exists
string Name
string FullName
string Extension
DirectoryInfo Parent
DirectoryInfo Root

DirectoryInfo Static Methods
bool Exists(string s)
DirectoryInfo GetParent(string s)
string GetDirectoryRoot(string s)

DirectoryInfo Methods
void Create()
DirectoryInfo CreateSubdirectory(string s)
void Refresh()

Directory Static Methods
DirectoryInfo CreateDirectory(string s)

DirectoryInfo Delete Methods
void Delete()
void Delete(bool recursive)

Directory Static Delete Methods
void Delete(string s)
void Delete(string s, bool recursive)

DirectoryInfo Properties
FileAttributes Attributes
DateTime CreationTime
DateTime LastAccessTime
DateTime LastWriteTime

Directory Static Methods
DateTime GetCreationTime(string s)
DateTime GetLastAccessTime(string s)
DateTime GetLastWriteTime(string s)
void SetCreationTime(string s, DateTime dt)
void SetLastAccessTime(string s, DateTime dt)
void SetLastWriteTime(string s, DateTime dt)

FileAttributes Enumeration

DirectoryInfo Methods
void MoveTo(string s)

Directory Static Methods
void Move(string src, string dst)

DirectoryInfo Methods
DirectoryInfo[] GetDirectories()
DirectoryInfo[] GetDirectories(string pattern)
FileInfo[] GetFiles()
FileInfo[] GetFiles(string pattern)
FileSystemInfo[] GetFileSystemInfos()
FileSystemInfo[] GetFileSystemInfos(string pattern)

Directory Static Methods
string[] GetDirectories(string s)
string[] GetDirectories(string s, string pattern)
string[] GetFiles(string s)
string[] GetFiles(string s, string pattern)
string[] GetFileSystemEntries(string s)
string[] GetFileSystemEntries(string s, string pattern)

File Manipulation

- File and FileInfo classes are similar to Directory and DirectoryInfo classes
  - File: **static** methods
  - FileInfo: **non-static** methods
- Two similar ways to get the same information
- Both are **sealed** (can't be subclassed) and derive from abstract class **FileSystemInfo**

Exercise

- Build a GUI with a TreeView control to display directories and files rooted at the current directory.
- Use a SplitContainer to place the tree on one side and a TextBox on the other.