

.NET Windows Forms Development and Deployment

What to Expect

- **Programming with Windows Forms**
- **Building Controls**
- **Multithreading and Windows Forms**
- **Code Access Security in Smart Client Applications**
- **Deploying Smart Client Applications**



Windows Forms

Topics

- **Comparing Win32 and Windows Forms**
- **Windows Forms programming model**
- **Windows Forms standard classes and controls**
- **Visual Studio .NET integration**



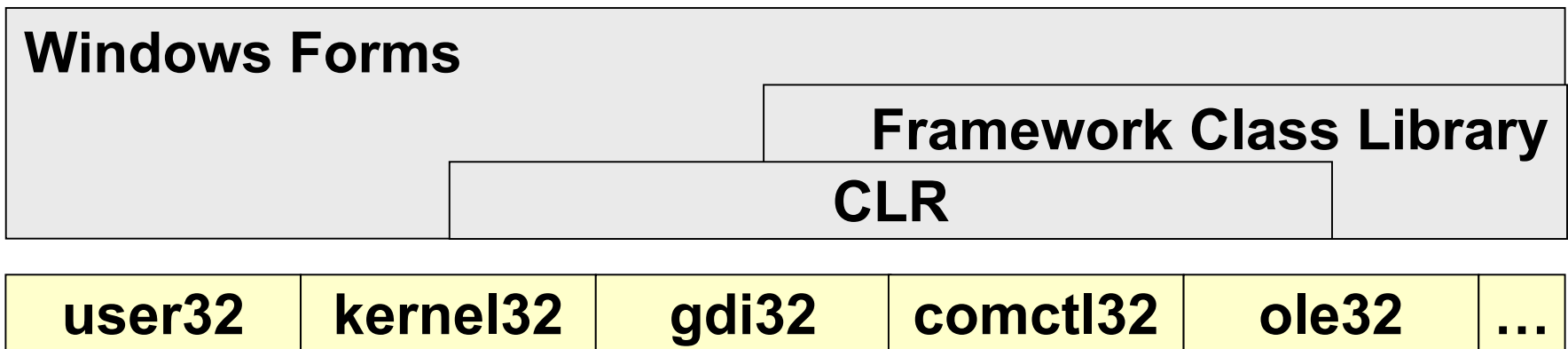
What is Windows Forms (a.k.a. WinForms)?

- **Windows Forms is part of the .NET framework**
 - Core classes in System.Windows.Forms namespace
 - Design-time support in various namespaces
- **Windows Forms provides classes for building UIs**
 - E.g. custom forms, common controls, standard dialogs
- **Visual Studio .NET provides tools for using Windows Forms**
 - Templates for common starting places, and a visual designer
- **GDI+ adds drawing and printing support to Windows Forms**
- **Windows Forms is not an application framework**
 - .NET itself provides a great deal of functionality outside of Windows Forms
 - Document-based applications not directly supported



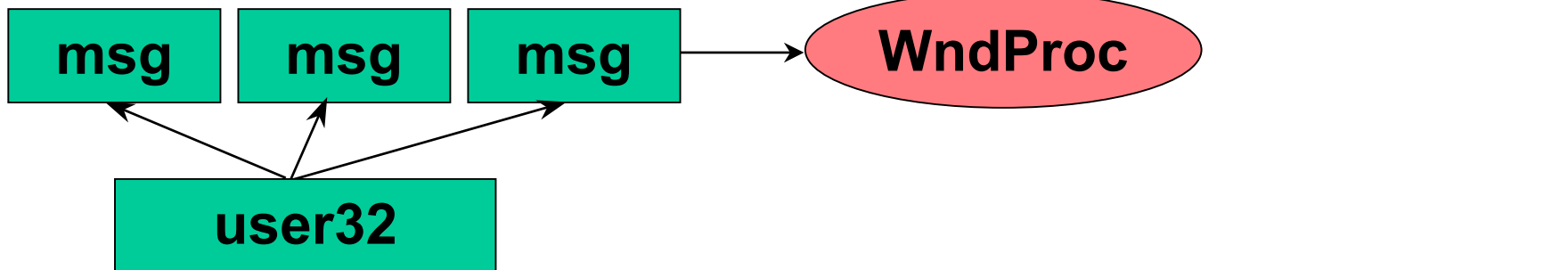
The Windows Forms Abstraction

- **Windows Forms is a layer of abstraction on top of the underlying Win32 windowing API (User32 et al)**
 - Theoretically allows Windows Forms to be implemented on other platforms, e.g. WinCE, Linux, FreeBSD
 - In practice, Win32 sometimes leaks through
- **Useful to understand how the mapping works when...**
 - Debugging
 - Integrating with unmanaged code (especially COM)



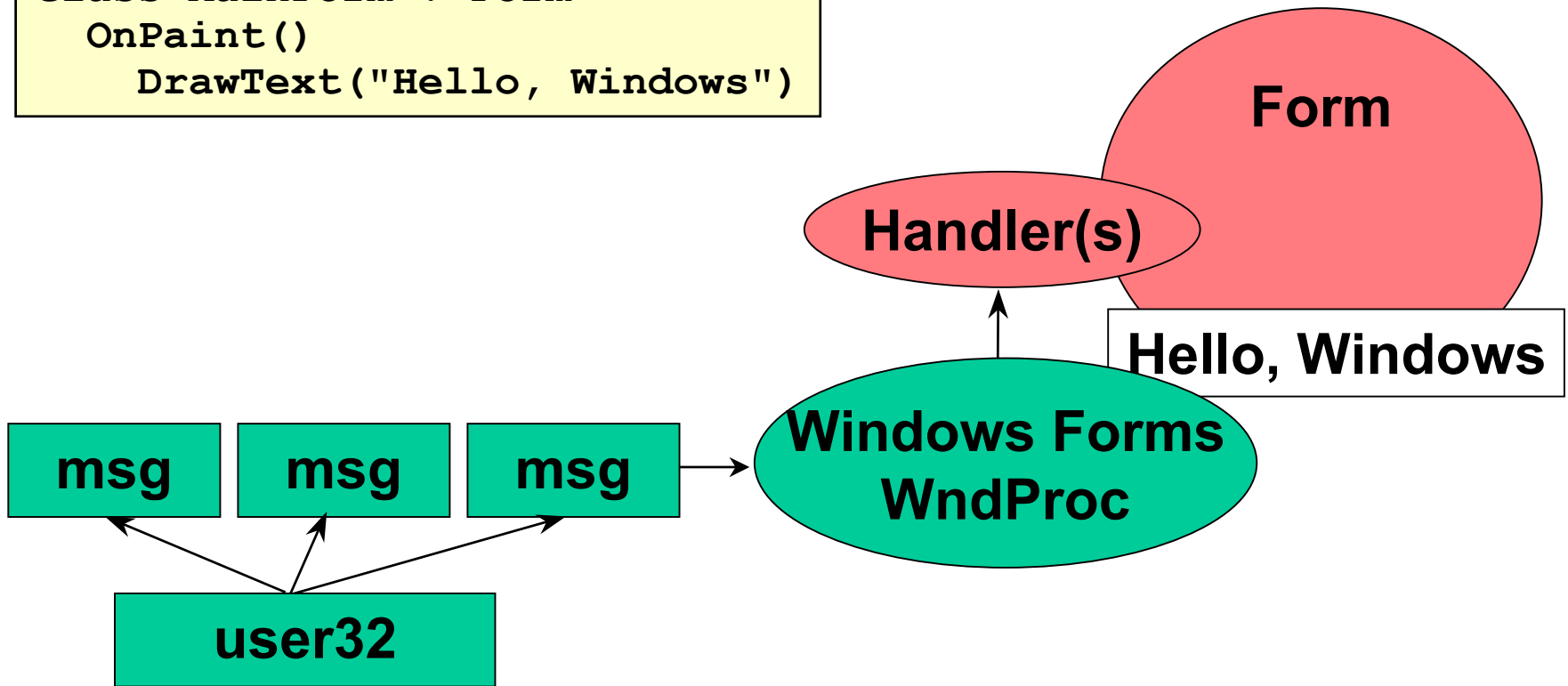
Anatomy of a Win32 Program

```
WinMain()  
  RegisterClassEx("MyMainWindowClass", WndProc)  
  hwnd = CreateWindow("MyMainWindowClass")  
  ShowWindow(hwnd)  
  while (GetMessage(&msg)) DispatchMessage(&msg)  
  
WndProc(message)  
  switch( message )  
    case WM_PAINT: OnPaint()  
  
OnPaint()  
  DrawText("Hello, Windows")
```



Anatomy of a Windows Forms Program

```
Main()  
    Form form = new MainForm()  
    Application.Run(form)  
  
class MainForm : Form  
    OnPaint()  
        DrawText("Hello, Windows")
```



Anatomy of a Windows Program

- Any Windows program has several common features
- Windows Forms provides the mapping from Win32

Feature	Win32	Windows Forms
Program Entry Point	WinMain	Main
Window Types	WNDCLASS	Form (or derived class)
Windows	HWND	Instance of Form
User Input Mechanism	Message Queue	Application.Run
Responding to Input	WndProc	Event Handlers



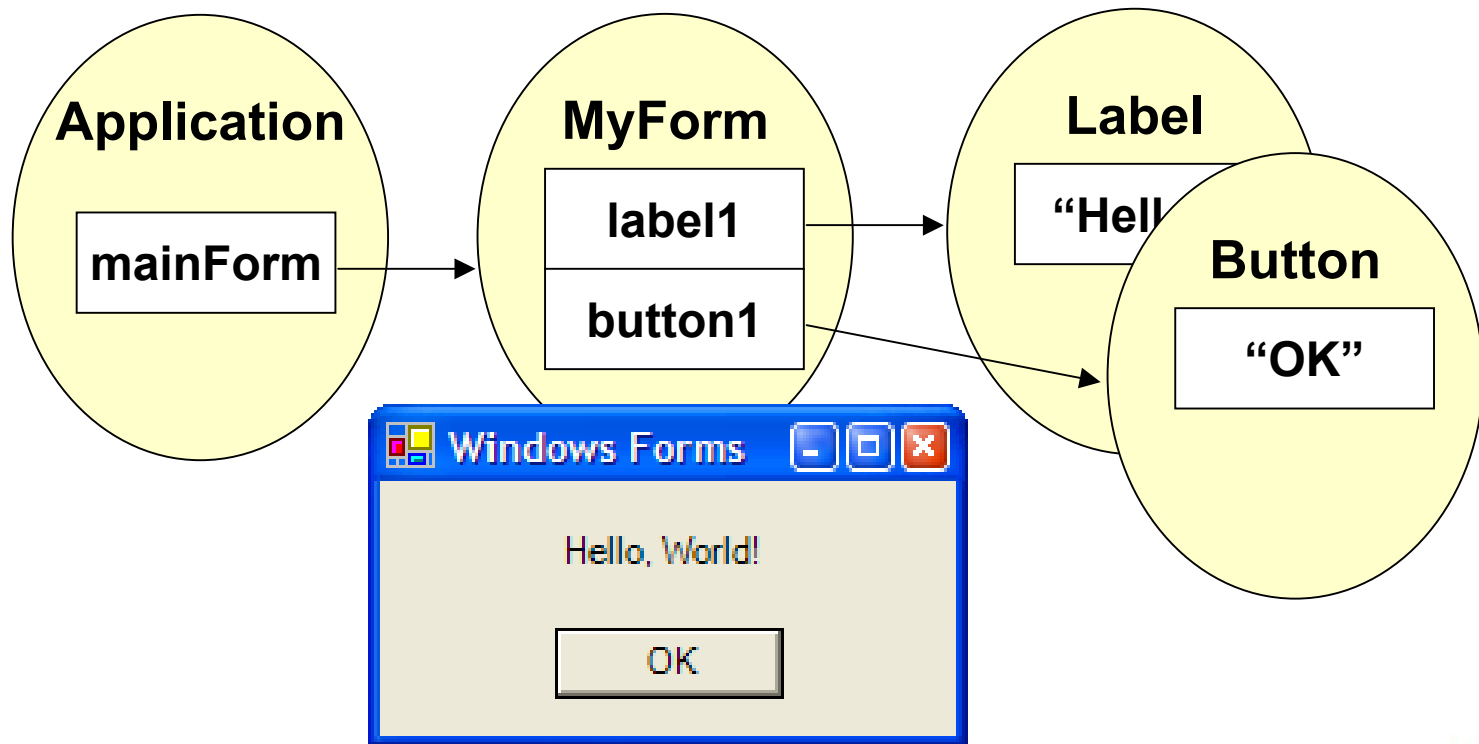
What is Windows Forms Good For?

- **Windows Forms is a good choice any time you need a Windows application**
 - Easier to use than MFC, WTL or Win32
 - More powerful than VB6
 - More UI capability than HTML
(with the same ease of deployment)
- **Possible Windows Forms uses include:**
 - N-tier clients
 - Web services front-ends
 - Intranet “smart clients”
 - Stand-alone Windows applications



Windows Forms Application Structure

- **A Windows Forms application has three pieces**
 - The application itself
 - Forms in the application
 - Controls on the form



System.Windows.Forms.Application

- **The Application class represents the application itself**
 - No instances (all properties and methods are static)
 - Processes UI events delivered by Windows
 - Run, DoEvents
 - Provides access to application environment
 - ExecutablePath, StartupPath
 - CommonAppDataPath, UserAppDataPath
 - CommonAppDataRegistry, UserAppDataRegistry

```
class MyApp {  
    public static void Main() {  
        MyForm form = new MyForm();  
        System.Windows.Forms.Application.Run(form);  
    }  
}
```



System.Windows.Forms.Form

- **Instances of the Form class represent windows**
 - Provide window-style services, e.g.
 - properties: Text, Size, Location, Controls
 - methods: Show, ShowDialog, Close
 - events: Load, Click, Closing
 - Custom forms typically derive from base Form class

```
class MyForm : Form {  
    public MyForm() {  
        this.Text = "This is my form!";  
        this.Location = new Point(10, 10);  
        this.Size = new Size(100, 100);  
    }  
}
```



Controls

- **Controls are visual components**
 - System.Windows.Forms.Control is base class for UI elements
 - e.g. Form, Button, Label, TextBox, ListBox, etc.
 - Contained and arranged by parent (usually a Form)
 - Held in parent's Controls collection
 - Forms usually handle events of child controls

```
public MyForm() {  
    Button button = new Button();  
    button.Text = "Click Me!";  
    button.Location = new Point(10, 10);  
    button.Click += new EventHandler(button_Click);  
    this.Controls.Add(button);  
}  
  
void button_Click(object sender, EventArgs e) {  
    MessageBox.Show("Thanks!");  
}
```



Windows Forms classes

- **System.Windows.Forms namespace**
 - Standard controls (e.g., Button, TextBox, TreeView etc.)
 - Base classes for custom controls (Control, ScrollableControl, ContainerControl, UserControl)
 - Event handling classes (XxxHandler, XxxEventArgs)
- **System.Drawing and children (aka GDI+)**
 - Wide range of drawing and composition primitives
 - Image handling and processing
 - Printing
- **Designer integration namespaces:**
 - System.ComponentModel (generic design-time support)
 - System.Windows.Forms.Design



Event Handling

- **.NET delegates have a specific pattern**
 - Delegate named with EventHandler suffix
 - First argument is event source (e.g., a control)
 - Second argument is EventArgs or derived
 - Returns void (any output data is set in EventArgs)
- **Handlers attached via class members called 'events'**

```
public delegate void MouseEventHandler(  
    object sender,  
    MouseEventArgs e);
```

```
class MouseEventArgs : EventArgs {  
    public MouseButton button;  
    public int clicks;  
    public int delta;  
    public int x;  
    public int y;  
}
```

```
public event MouseEventHandler MouseMove;
```



Events and Overridable Methods

- Most events have corresponding virtual methods

```
public delegate void MouseEventHandler(  
    object sender, MouseEventArgs e);  
  
public class EventSource {  
    public event MouseEventHandler MouseMove;  
    ...  
    protected virtual void OnMouseMove(  
        MouseEventArgs e) { ... }  
}
```

- Object users must subscribe to events
- Derived classes can override virtual methods
 - Must call base class to fire events to subscribers

```
protected override void OnMouseMove(MouseEventArgs e) {  
    base.OnMouseMove(e);  
    // something interesting here...  
}
```



Handling Events vs. Overriding Methods

- **Overriding allows control over order of event handling**

```
protected override void OnMouseMove (MouseEventArgs e) {  
    ...do something before event handlers...  
  
    base.OnMouseMove (e) ;  
  
    ...do something after event handlers...  
}
```

- **Overriding very marginally more efficient**
 - Virtual function call slightly faster than call through delegate
 - Slightly less memory required per instance
- **Efficiency gains usually irrelevant in practice**
 - Control over order is the only real reason for overriding



Windows Forms Project Templates in VS.NET

- **Project templates create small starter projects containing:**
 - App.ico: application's icon
 - AssemblyInfo.cs: assembly-level attributes
 - Form1.cs: main form and Main entry point
- **Wizard-produced project references several assemblies:**
 - System, System.Data, System.Drawing, System.Windows.Forms, System.XML
- **Individual Form templates provide support for the Designer**



Visual Studio .NET Forms Designer

- **Visual environment for building Forms and Controls**
- **Drag-and-drop editing of child controls**
- **Property grid to configure both form and controls:**
 - Setting form and control properties
 - Handling form and control events (but not in VB.NET)
- **Also supports non-visual components**
 - Appear in 'component tray' at design time
 - Can be edited with property grid just like controls
 - Do not appear at all at runtime



Designer Code Generation

- **The Forms Designer generates code in InitializeComponent**
 - Creates and initializes child controls and components
 - Must be called from constructor
 - Regenerated any time a change is made with the Designer

```
public MyForm() {
    InitializeComponent();
    // your stuff here...
}

private void InitializeComponent() {
    this.button1 = new System.Windows.Forms.Button();
    ...
    this.SuspendLayout();
    ...
    this.Controls.AddRange(new Control[] { this.button1, ... });
    this.Name = "MyForm";
    this.Text = "My Form!";
    this.ResumeLayout(false);
}
```

Summary

- **Windows Forms is a managed layer on top of Win32**
- **UI elements derive from the Control base class**
- **Top-level windows derive from Form base class**
- **Events from OS managed by Application class**
- **VS.NET provides simple templates as starting point**
- **VS.NET Forms Designer works by generating code**



Controls

Topics

- **Built-In Controls and Components**
- **Custom Components**
- **Tool Integration**
- **Custom Controls**
- **Rendering and Input**
- **Scrollable Controls**
- **User Controls**
- **COM Controls**

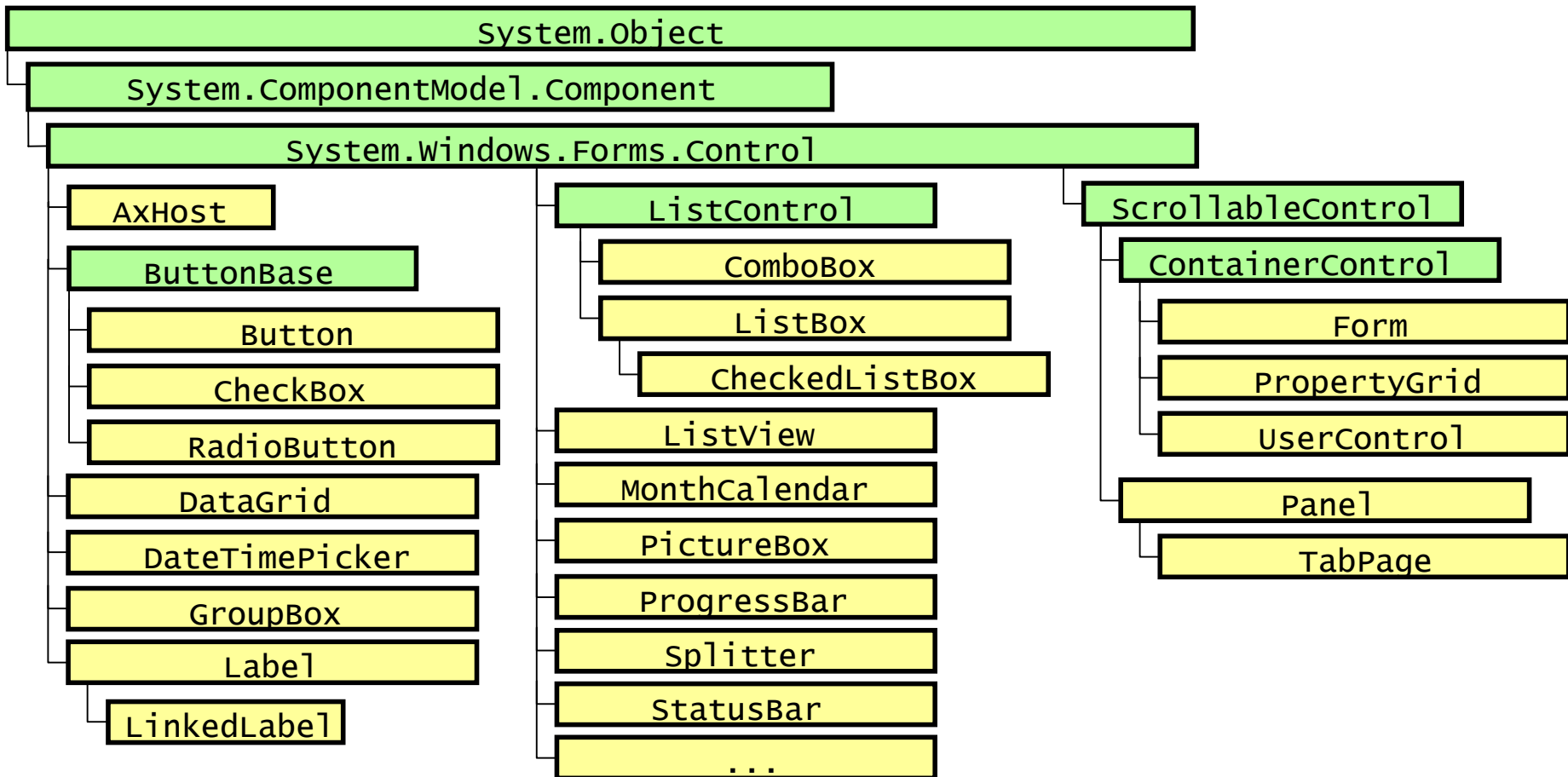


Windows Forms Controls

- **Controls are objects that are visible at runtime**
 - Manage their own appearance
 - Usually support interaction with user
- **Windows Forms provides many built in control classes**
 - Equivalents to Win32 controls, e.g. Button, ListBox, TreeView
 - Some new to Windows Forms, e.g. PropertyGrid, DataGrid
- **Can write custom control classes**
 - From scratch (derive from Control)
 - With Forms Designer (derive from UserControl)
 - Based on existing control (either use UserControl with containment, or derive from existing control)



Built-In Controls



Components

- **Components are classes with design-time support**
 - Drag and drop from the Toolbox
 - Show up on component tray
 - Set properties via the Property Browser
 - Control class derives from Component
 - Do not need to be a control just to integrate with VS.NET
- **Some built-in components oriented towards client-side apps**
 - HelpProvider, ImageList, Timer, standard dialogs, ErrorProvider, ToolTip, NotifyIcon
- **Some towards components and server-side apps**
 - FileSystemWatcher, EventLog, DirectoryEntry, DirectorySearcher, MessageQueue, PerformanceCounter, Process, ServiceController



Component Tool Integration

- **Designer provides drag-n-drop environment for components**
 - Components show on tray, configurable via Property Browser
 - Designer writes code to set properties in InitializeComponent

```
void InitializeComponent() {  
    ...  
    conn.ServerName = "mydb.com";  
    conn.DatabaseName = "pubs";  
    ...  
}
```



Customizing Component Integration

- **System.ComponentModel** provides attributes for customizing tool integration
 - E.g. Category, Description, and DefaultValue
- **Not used at runtime – just there to help the VS.NET Designer**
 - Designer uses Category and Description in PropertyGrid
 - Designer only writes code to set non-default values

```
[Description("Name of server")]  
[DefaultValue("localhost")]  
string ServerName {  
    get { return _serverName; }  
    set { _serverName = value; }  
}
```



Design Mode

- **Some components behave differently in the designer**
 - E.g. not actually making connection to a database
- **DesignMode property set by Designer after construction**
 - It will always be false in the ctor
 - Implement ISupportInitialize to check during initialization

```
bool Connect() {  
    // Act differently during design mode  
    if( DesignMode ) return true;  
  
    ... // At runtime, do actual work  
}
```



Custom Controls

- **All controls are components**
 - Control derives from `System.ComponentModel.Component`
 - Designer integration attributes therefore supported
- **Custom controls derive from `System.Windows.Forms.Control`**
 - Standard component features therefore free
 - Designer support
 - Resource management
 - Control class also adds lots of properties, methods and events
- **Custom controls extend base `Control` functionality**
 - Custom properties, methods and events
 - Provide output via rendering
 - Consume input via mouse and keyboard



Control Rendering

- Custom controls render their state by handling Paint event

```
// A complete custom control
public class MyLabelControl : Control {
    protected override void OnPaint(PaintEventArgs pe) {

        // Render our state using members from base Control
        using (Brush brush = new SolidBrush(this.ForeColor)) {
            pe.Graphics.DrawString(this.Text, this.Font, brush,
                                   0, 0);
        }

        // Notify listeners last
        base.OnPaint(pe);
    }
}
```



Control Re-Rendering

- When a control's state changes, it needs to request a redraw
 - Invalidate will cause a Paint event

```
class MyLabelControl : Control {
    override void OnTextChanged(EventArgs e) {
        base.OnTextChanged(e);
        Invalidate();
    }

    override void OnFontChanged(EventArgs e) {
        base.OnFontChanged(e);
        Invalidate();
    }
    ... // also need to watch back/forecolor
}
```



Control Input

- **Users interact directly with a control via mouse and keyboard**
 - Users interact indirectly via activities that cause container to call methods, set properties, etc.
- **Can handle input by handling the mouse or keyboard events**
 - Default implementations fire the event for the container

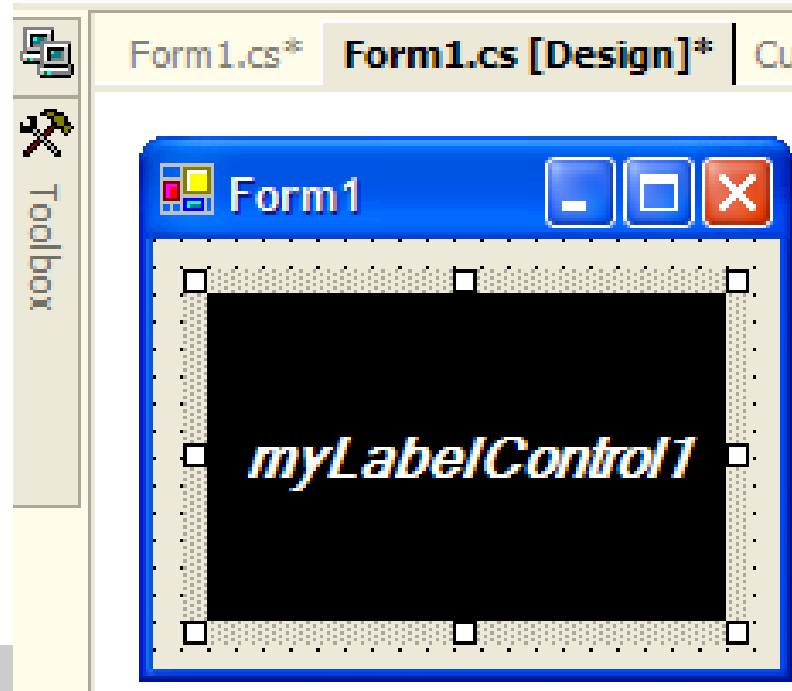
```
// Mouse Events  
OnMouseDown (MouseEventArgs e)  
OnMouseEnter (EventArgs e)  
OnMouseHover (EventArgs e)  
OnMouseLeave (EventArgs e)  
OnMouseMove (MouseEventArgs e)  
OnMouseUp (MouseEventArgs e)  
OnMouseWheel (MouseEventArgs e)
```

```
// Keyboard Events  
OnKeyDown (KeyEventArgs e)  
OnKeyPress (KeyPressEventArgs e)  
OnKeyUp (KeyEventArgs e)
```



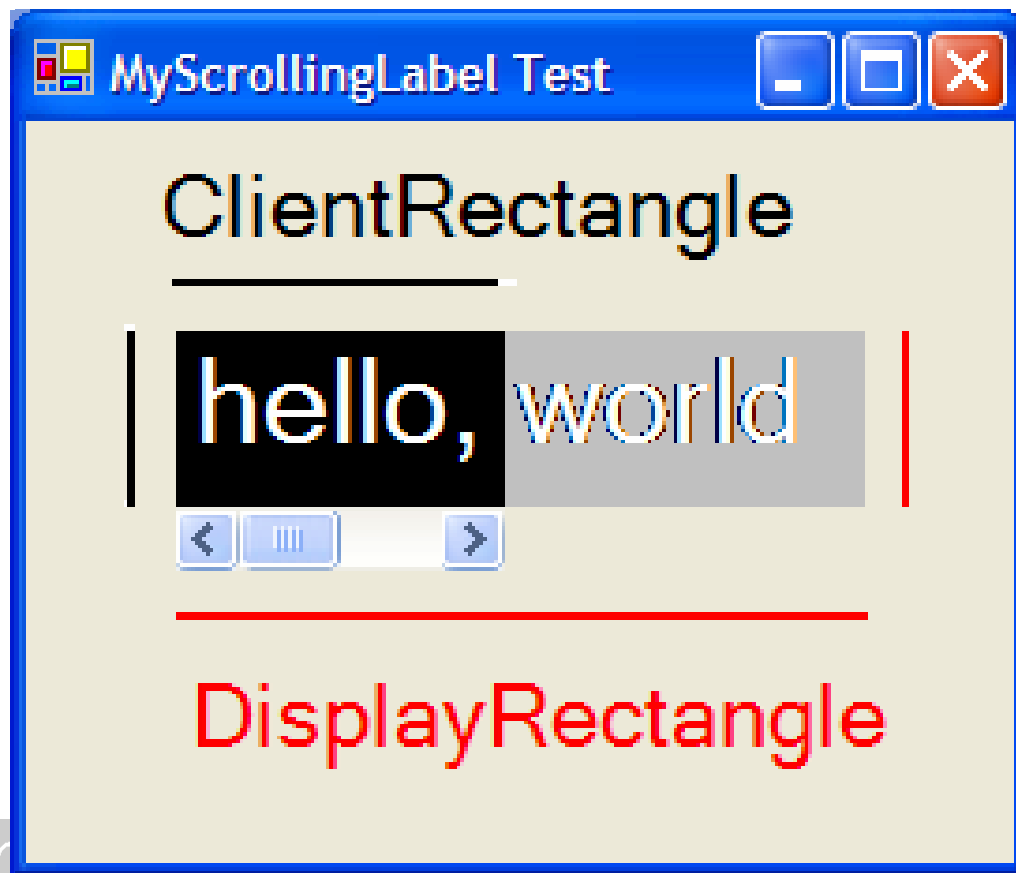
Tool Integration

- **Controls get all of Component integration**
 - Drag-n-drop, PropertyGrid support, custom attributes
- **Controls are rendered in their container in design mode**
 - Can have custom controls in your application project
 - Designer shows last compiled version



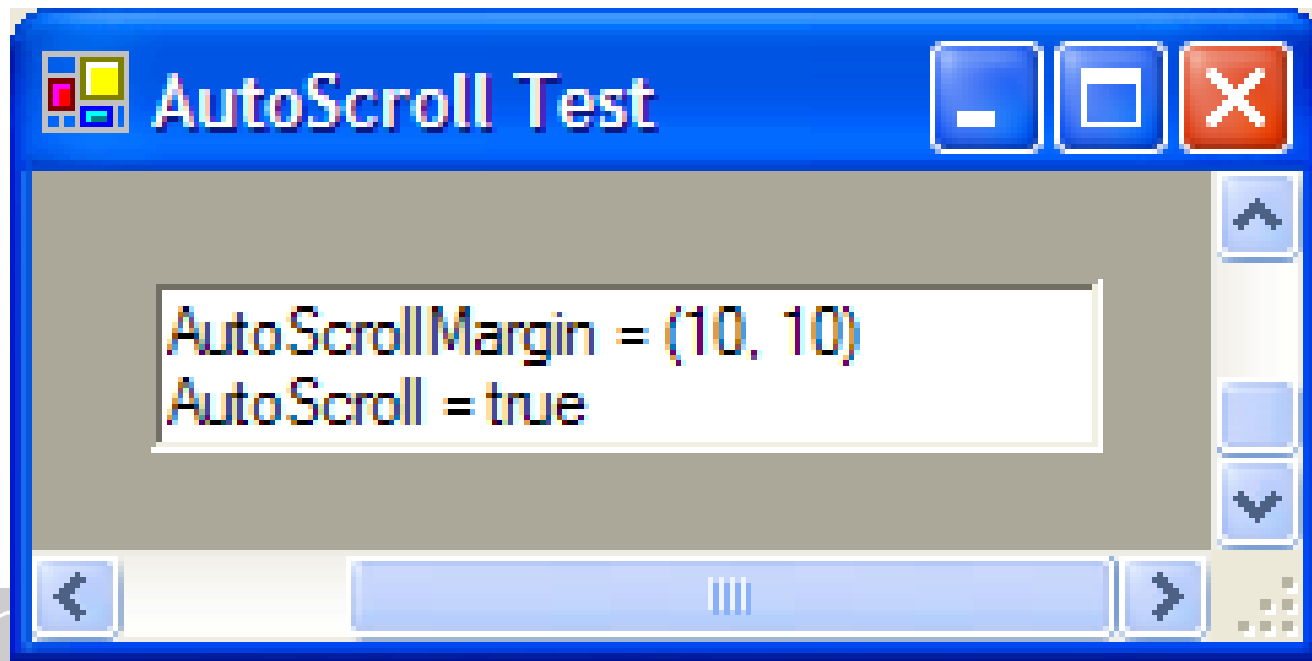
Scrollable Controls

- **ScrollableControl-derived controls have optional scrollbars**
 - ClientRectangle is visible drawing region
 - DisplayRectangle is entire drawing region, including invisible



Scrollable Control Area

- **DisplayRectangle set indirectly**
 - AutoScrollMinSize is base size
 - Ignored if smaller than ClientRectangle
 - Bottom right edge of child controls inflate DisplayRectangle
 - Only if AutoScroll is set to true
 - AutoScrollMargin pad child controls



Ex: ScrollableControl

```
public class MyScrollingLabel : ScrollableControl {
    protected override void OnPaint(PaintEventArgs pe) {
        // Paint using DisplayRectangle instead of ClientRectangle
        pe.Graphics.DrawString(Text, Font, Brushes.Black,
                               DisplayRectangle);
    }

    protected override void OnTextChanged(EventArgs e) {
        base.OnTextChanged(e);
        Invalidate();
        SetScrollMinSize();
    }
    ...
    void SetScrollMinSize() {
        using (Graphics g = this.CreateGraphics()) {
            SizeF sizeF = g.MeasureString(this.Text, this.Font);
            Size size = new Size((int)Math.Ceiling(sizeF.Width),
                                (int)Math.Ceiling(sizeF.Height));
            this.AutoScrollMinSize = size;
        }
    }
}
```

Derived Controls

- **All controls ultimately derive from Control class**
 - But it's OK to derive indirectly
- **Can derive from built-in controls**
- **Adding custom drawing may present issues**
 - Some controls repainted by OS (e.g., TextBox)
 - OnPaint not necessarily called
- **Inheritance not the only way to reuse controls**
 - Reuse through containment may be better
 - Just place in a UserControl and set Dock to Fill



Containment vs. Derivation

- **Containment offers similar advantages to derivation**
 - Control looks and feels to user just like original control
 - Reuse of code – no need to reinvent wheel
- **Permits better encapsulation**
 - Can select exactly which features of the original to make public
- **Derivation still has its uses**
 - New control type-compatible with base (may be good or bad)



Visual Inheritance

- **Visual Studio .NET has built-in support for inheritance**
 - Allows visual editing of certain derived UI elements
- **Derived Forms**
 - Useful for apps with many similar windows
- **Derived UserControls**
 - Less useful (available because Forms and UserControls are treated in very similar fashion by VS.NET)
- **UserControl offers non-derivation-based alternative**
 - Although inheritance allows derived control to modify the base control



User Controls

- **User Controls are small reusable pieces of UI**
 - User Control encapsulates both appearance and behavior
- **VS.NET treats them like forms without the frame**
 - Derives from `System.Windows.Forms.UserControl`
 - `System.Windows.Forms.Form` shares the same base class
 - `System.Windows.Forms.ContainerControl`
 - Work in exactly the same way in the designer
- **User Controls contain other controls**
 - Useful for gathering other controls together
 - Designer provides Form-like drag-n-drop environment
 - Control renders itself just like a Form



ActiveX Controls

- **VS.NET can import COM Controls**
 - Can use aximp.exe command line tool
- **Creates a new class derived from AxHost**
 - Special code to host a COM Control
- **New class behaves like a Windows Forms control**
 - COM methods, properties and events mapped to .NET
- **COM Controls need to live in a single-thread COM apartment**
 - .NET will load COM on demand, assuming multi-threaded apt.
 - Need to mark a thread's entry point with attribute to set STA

```
[STAThread]
static void Main() {
    Application.Run(new Form1());
}
```



Summary

- **.NET supplies many built-in controls and components**
- **Tool integration available for both controls and components**
- **Custom controls derive from Control**
- **Scrolling with ScrollableControl**
- **Can derive from other controls**
- **User controls often best means of UI reuse**
- **COM control support available**



Multithreading

Responsiveness

- **Crucial for perceived quality**
- **Mustn't freeze user interface**
- **Allow cancellation**
- **Provide feedback**
- **Multithreaded code usually required**



Responsibility

- **Multithreaded code is *HARD***
- **Most components not thread safe**
 - Including most of the .NET FCL
- **Must therefore deal with concurrency**
- **Windows Forms has thread affinity**
- **Must therefore avoid deadlock and other subtle bugs**



The Golden Rule

Only use a Control on
the thread on which it
was created



Keeping the UI Responsive

- **Don't block UI thread**
- **Many operations block**
 - Most IO (that includes the filesystem!)
 - Web Services
- **Non-IO operations can be slow too**
 - E.g. Image processing, sound compression
- **Do all such work asynchronously**



Asynchronous Options

- **Create new thread**
- **Use the Thread Pool**
 - Async Delegate Invocation (best)
 - Async Socket programming
 - System.Timers.Timer
 - System.Threading.Timer
 - System.Threading.ThreadPool



Use the Thread Pool

- **CLR Thread Pool == Good Stuff**
 - Creates and destroys threads as required
 - Balances thread count against system load
 - Transfers thread security info correctly
- **Async Delegate Invocation simplest**
 - Pass arbitrary parameters (even out or ref)
 - Detect completion through poll or callback



Async Delegate Invocation

- **Choose or define appropriate delegate**

```
public delegate void MyDelegate(string param);
```

- **Point delegate at function**

```
private void SomeFunction(string s) { ... }  
MyDelegate dlg = new MyDelegate(SomeFunction);
```

- **Use BeginInvoke**

```
dlg.BeginInvoke("Hello, world", null, null);
```

- **That is all**



Async Delegate Example

```
// Slow IO-bound method
public void FetchDataSync(string url, string savePath)
{
    WebClient webClient = new WebClient();
    webClient.DownloadFile(url, savePath);
}

// Delegate to wrap slow method
public delegate void FetchDelegate(string url,
                                   string savePath);

// Call slow method asynchronously
public void FetchDataAsync(string url, string savePath)
{
    FetchDelegate d1g = new FetchDelegate(FetchDataSync);
    d1g.BeginInvoke(url, savePath, null, null);
}
```



Detecting Completion

- **Fire and forget not always appropriate**
- **Can poll for completion**
 - Usually simplest, but not most efficient
- **Can get completion callback**
 - More efficient
 - Likely to raise harder threading issues



Polling

- **Check regularly to see if done**
- **Use `System.Windows.Forms.Timer`**
- **Async Delegates return `IAsyncResult`**
 - Can check `IsCompleted`
- **Could also use ad hoc flag**
 - May be useful if not using delegates
 - Care needed to ensure thread safety



Polling Example

```
private IAsyncResult callObject;
private void FetchAsyncPolling(string url, string savePath)
{
    FetchDelegate dlg = new FetchDelegate(FetchDataSync);
    callObject = dlg.BeginInvoke(url, savePath,
                                null, null);

    pollTimer.Enabled = true;
}

private void pollTimer_Tick(object sender, EventArgs e)
{
    if (callObject != null && callObject.IsCompleted)
    {
        lblStatus.Text = "Done";
        pollTimer.Enabled = false;
    }
}
```



Event-based Notification

- **Get notified as soon as work done**
- **For Async delegates, use AsyncCallback**
 - Provide a second delegate
 - Passed to BeginInvoke
 - Will be called when work done
 - Callback happens on a Thread Pool thread!
- **(This is how Async Sockets always work)**



Async Delegate with Callback

```
private void AsyncwithCallback (string url, string savePath)
{
    FetchDelegate d1g = new FetchDelegate(FetchDataSync);
    d1g.BeginInvoke(url, savePath,
                   new AsyncCallback(OnFetchComplete), null);
}

private void OnFetchComplete(IAAsyncResult ar)
{
    MessageBox.Show("Done", "Fetch");

    // Mustn't do this - on wrong thread!
    // lblStatus.Text = "Done";
}
```



The Golden Rule (again)

Only use a Control on
the thread on which it
was created



Notification and Windows Forms

- **Async callback is on Thread Pool thread**
- **Can't do anything to UI on this thread**
- **Need to marshal call onto UI thread**
- **ISynchronizeInvoke is your friend**



ISynchronizeInvoke

- **Marshals calls to known context**
- **Implemented by Control**
 - Marshals calls to UI Thread
- **Can detect whether marshaling required**
 - InvokeRequired
- **Supports synchronous and asynchronous**
 - Invoke, or BeginInvoke and EndInvoke



ISynchronizeInvoke Example

```
private void AsyncwithCallback (string url, string savePath)
{
    FetchDelegate d1g = new FetchDelegate(FetchDataSync);
    d1g.BeginInvoke(url, savePath,
                   new AsyncCallback(OnFetchComplete), null);
}

private void OnFetchComplete(IAAsyncResult ar)
{
    if (InvokeRequired)
    {
        AsyncCallback cb = new AsyncCallback(OnFetchComplete);
        BeginInvoke(cb, new object[] { ar } );
        return;
    }

    lblStatus.Text = "Done";
}
```



Invoke and Deadlock

- **ISynchronizeInvoke.Invoke is synchronous**
- **Control's Invoke implementation**
 - Will block until UI thread is available
 - Could therefore deadlock if UI thread blocked
- **Always prefer BeginInvoke**



Example Deadlock

```
private Queue queue = new Queue();
private void DoWorkOnWorkerThread()
{
    lock (queue)
    {
        queue.Enqueue("Data!");
        MethodInvocation dlg = new MethodInvocation(RunOnUIThread);
        Invoke(dlg, null); // Deadlock! Should use BeginInvoke
    }
}

private void RunOnUIThread()
{
    string s;
    lock (queue) // will deadlock here
    {
        s = queue.Dequeue().ToString();
    }
    lblOutput.Text = s;
}
```



Avoiding Concurrency

- **Multithreaded code is hard**
- **Complexity mostly due to concurrency**
 - Need to protect shared state with locks
 - Locking strategy must avoid deadlocks
 - Fragile—one bad apple spoils whole cart
- **Best to avoid concurrency entirely**
 - Use message passing idiom



Message Passing

- **Multithreading in UI is for responsiveness**
 - Not about performance on SMP systems!
 - Concurrency therefore strictly optional
- **Any object owned by exactly one thread**
- **Ownership can change over time**
 - Passed to worker thread at async invocation
 - Returned to UI thread by ISynchronizeInvoke



If Concurrency Is Unavoidable

- **Sometimes data must be shared**
- **Use built-in object Monitor**

```
lock(mySharedObject) {  
    . . . read or write shared state  
}
```
- **Lock for *all* reads and writes**
- **Keep it simple (preferably 1 lock at a time)**
- **Don't write your own transaction manager**



Dedicated Threads

- **CLR Thread Pool is usually great**
- **Mustn't hog threads**
- **For long-running work, create a thread**

```
Thread t;  
private void ThreadProc() { . . . }  
private LaunchThread() {  
    t = new Thread(new ThreadStart(ThreadProc));  
    t.Start();  
}
```



Status Updates

- **Supply feedback for slow operations**
 - Percentage progress where possible
 - Other information where suitable
- **Delays are less irritating when there is something to watch**
- **Use same techniques as for completion**
 - Poll (will need ad hoc technique)
 - Event-based (use `ISynchronizeInvoke`)



Cancellation

- **Lengthy operations must be cancelable**
- **With dedicated threads can use brute force**
 - Thread.Interrupt (or even Thread.Abort)
 - Not appropriate for Thread Pool threads
- **Usually better to have cancellation flag**
 - Worker thread checks flag periodically
 - Allows it to stop when it's ready



Asynchronous Cancellation

- **Remember cancellation may be slow**
- **Don't block UI waiting for cancellation**
- **Request, cancellation then wait**
 - Poll for cancellation
 - Raise cancellation event (using `ISynchronizeInvoke` of course)



Cancellation Example

```
private void btnGo_Click(object sender, EventArgs e) {
    btnGo.Enabled = false;    btnCancel.Enabled = true;
    MethodInvoker dlg = new MethodInvoker(work.DoWork);
    dlg.BeginInvoke(new AsyncCallback(OnComplete), null);
}

private void btnCancel_Click(object sender, EventArgs e) {
    btnCancel.Enabled = false;
    work.StartCancel();
}

private void OnComplete(IAsyncResult iar) {
    if (InvokeRequired) {
        AsyncCallback dlg = new AsyncCallback(OnComplete);
        BeginInvoke(dlg, new object[] { iar });
        return;
    }
    btnGo.Enabled = true;
}
```



Summary

- **High quality applications don't freeze**
- **Run slow operations asynchronously**
- **Use Thread Pool, via Async Delegates**
- **Avoid concurrency where possible**
- **Support feedback and cancellation**
- **Remember The Golden Rule**



The Golden Rule (yet again)

Only use a Control on
the thread on which it
was created



Code Access Security

Topics

- **The problem of mobile code**
- **Code Access Security (CAS)**
- **Evidence**
- **Permissions**
- **Policy**
- **Enforcement**
- **Tips for writing mobile code**



What is mobile code?

- **Mobile code**
 - you haven't explicitly installed it on your local machine
 - it comes to you via a network
 - it's often delivered via web pages
 - in the COM era, it was packaged as native code
 - in the .NET era, it is packaged as .NET assemblies
- **Two major sources of mobile code**
 - your company's intranet
 - the public Internet



The old model: Full Trust

- **In the COM era, mobile code was either fully trusted or completely untrusted**
- **User had to decide at download time**
 - user presented with certificate
 - asked whether they trusted the vendor or his code
- **NO usually means websites don't work**
 - mobile code not allowed to execute if you say no
- **YES could mean many things**
 - user gets a richer browsing experience
 - user gets a more buggy and vulnerable code base
 - user gets a virus or Trojan horse installed on his system

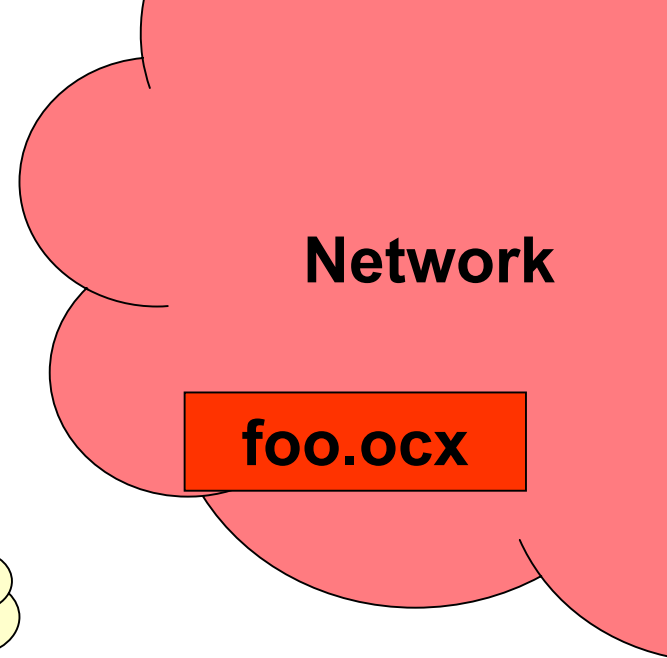
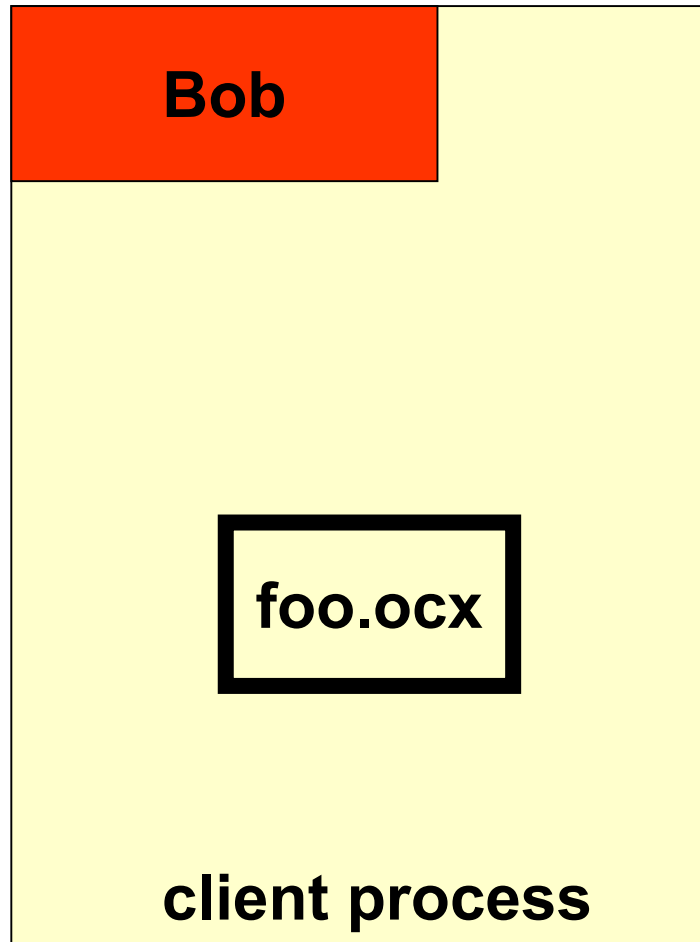


The COM loader

- **COM code is packaged in native DLLs**
- **Once loaded, a DLL becomes an integral part of the process**
 - practically speaking, you can't tell the difference between code in the DLL and code in the original application
 - every line of code runs with the same level of privilege



The COM loader, illustrated



Uh, sure, if I have to...



Do you trust foo.ocx to do anything you can do?

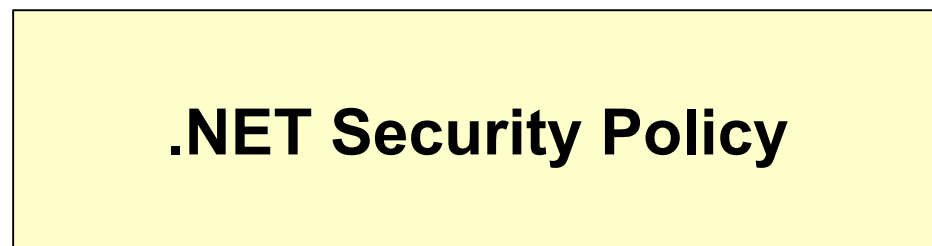
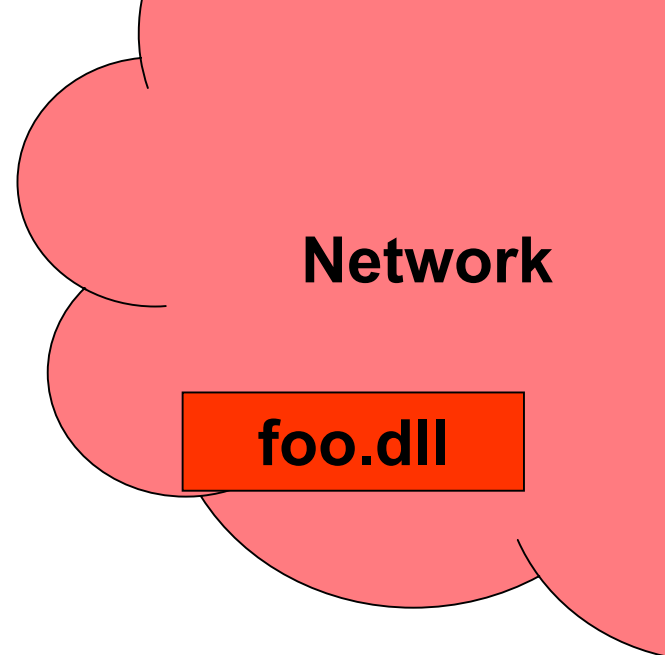
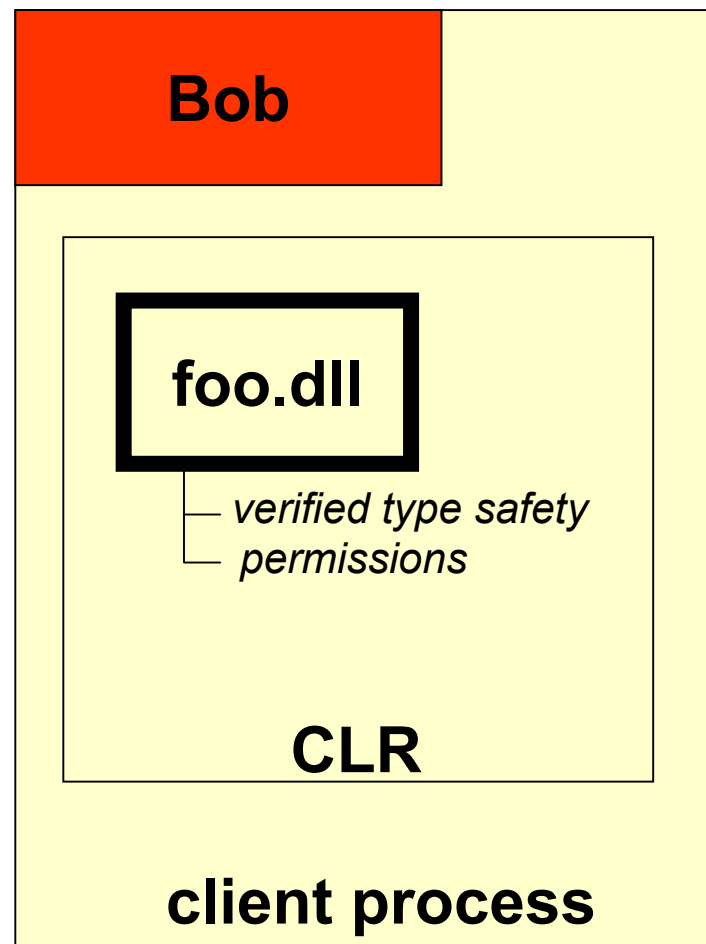


The .NET assembly loader

- **When assemblies are loaded, the loader gathers evidence**
 - where did this assembly come from?
 - what public key is this assembly signed with?
- **Evidence is pumped through .NET security policy**
 - result is a set of **permissions** for the assembly
- **Assembly runs with permissions assigned by policy**
 - mobile code likely won't be able to do everything the user can
- **User isn't asked trust questions they can't answer**
- **This feature is known as Code Access Security (CAS)**



The .NET loader, illustrated



The importance of type safety

- **The previous picture shows the boundary of foo.dll remains**
 - the system can constrain what mobile code can do
- **Only possible if the type system of the CLR is watertight**
- **This was impossible in older COM based systems**
 - C++ programmers could get around the type system with casts
 - VB6 programmers could do the same if they knew the right functions to call
- **This is the whole point to code verification**
 - ensures the code is verifiably type safe
 - a bug in the verification process could compromise the entire model



Evidence

- **Evidence can take many forms**
 - System.Security.Policy has several evidence classes
- **Where did this assembly come from?**
 - Zone evidence
 - Url evidence
 - Site evidence
 - ApplicationDirectory evidence
- **Who did this assembly come from?**
 - StrongName evidence
 - Publisher evidence (Authenticode cert)
- **What is this assembly?**
 - Hash evidence



Example: examining the evidence

```
using System;
using System.Reflection;
using System.Collections;

class App {
    static void Main(string[] args) {
        printEvidence(Assembly.Load(args[0]));
    }
    static void printEvidence(Assembly a) {
        Console.WriteLine("<evidence assembly='{0}'>",
            a.GetName().Name);
        IEnumerator it = a.Evidence.GetEnumerator();
        while (it.MoveNext()) {
            Console.WriteLine(it.Current);
        }
        Console.WriteLine("</evidence>");
    }
}
```



Permissions

- **Policy maps evidence to a set of permissions**
- **Permissions limit what an assembly can do**
 - might not be able to run if not verifiable
 - might not be able to call native code (COM objects, DLLs)
 - might not be able to access the file system at all
 - might be forced to use common dialogs to access files or printers, ensuring that the user has a say
 - might not be able to use certain environment variables
 - might not be able to use the network
- **Each permission is represented by a class**



CAS permission classes

```
DBDataPermission  
PrintingPermission  
MessageQueuePermission  
DnsPermission  
SocketPermission  
WebPermission  
EnvironmentPermission  
FileDialogPermission  
FileIOPermission  
IsolatedStorageFilePermission  
ReflectionPermission  
RegistryPermission  
SecurityPermission  
UIPermission
```



Policy maps evidence to permissions

- **Security policy says which assemblies get which permissions**
 - evidence is the input to policy
 - policy grants permissions based on the evidence
 - resulting permissions are assigned to assembly
- **Default policy is based on Internet Explorer security zones**
 - My Computer
 - Local Intranet
 - Trusted Sites
 - Restricted Sites
 - Internet



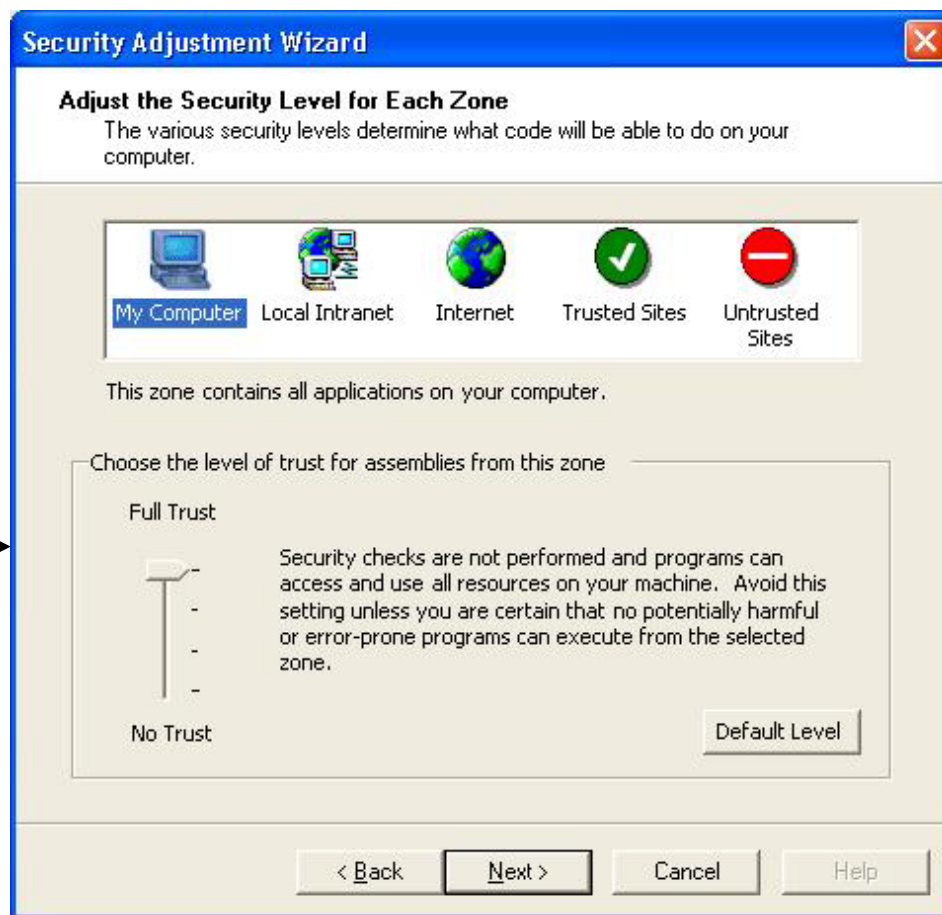
Permissions based on Zone (SP1 or greater)

My Computer	Full Trust (no limitations at all)
Local Intranet	Medium Trust
Trusted Sites	Low Trust
Restricted Sites	Nothing (cannot execute)
Internet	Nothing (cannot execute)



Using a wizard to adjust policy

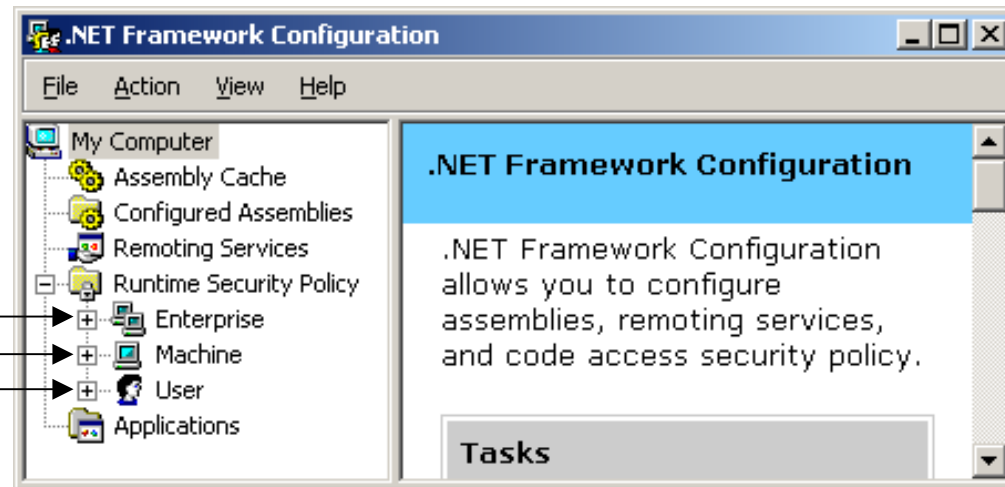
- Allows you to choose one of four permission sets
 - Full Trust
 - Medium Trust
 - Low Trust
 - No Trust



Drilling into security policy

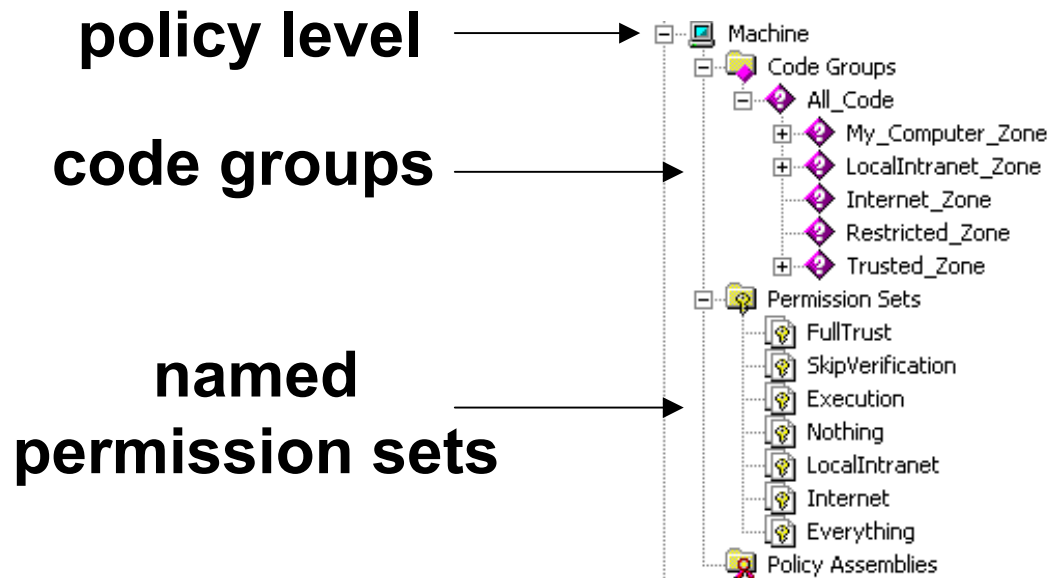
- **Policy is organized in several levels**
 - allows delegation of administration
 - restrictions at one level cannot be overridden in a lower level
- **View/edit using .NET Framework Configuration MMC snapin**

**policy
levels**



Drilling into a policy level

- A policy level consists of a tree of **code groups** and a list of **named permission sets**



Code groups

- **Each code group has two properties**
 - a membership condition (a single test of the evidence)
 - a reference to a named permission set
- **Permission grant for a policy level comes from the union of all matching code groups**
 - child nodes evaluated only if parent node matches



Evaluating a policy level

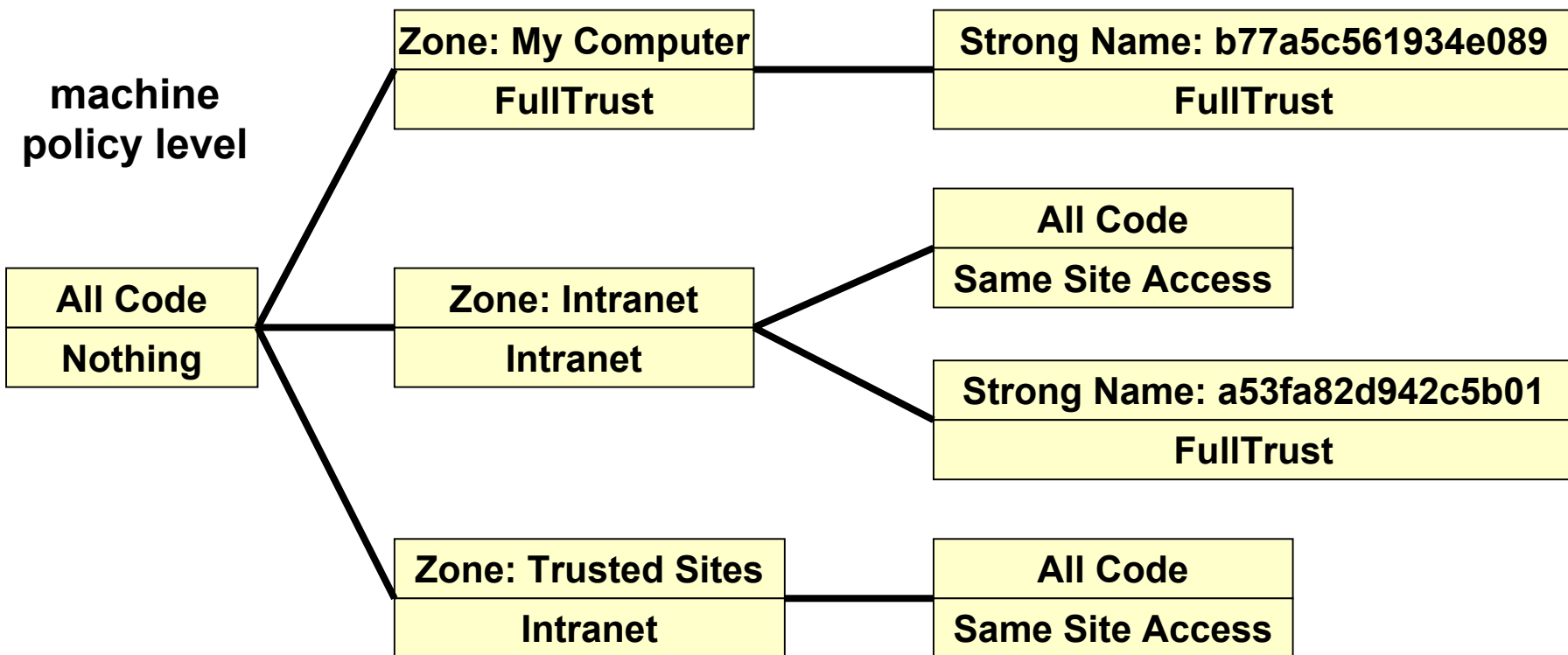
evidence:

URL: http://sales/routing.dll
Zone: Intranet
StrongName: a53fa82d942c5b01

permission grants:

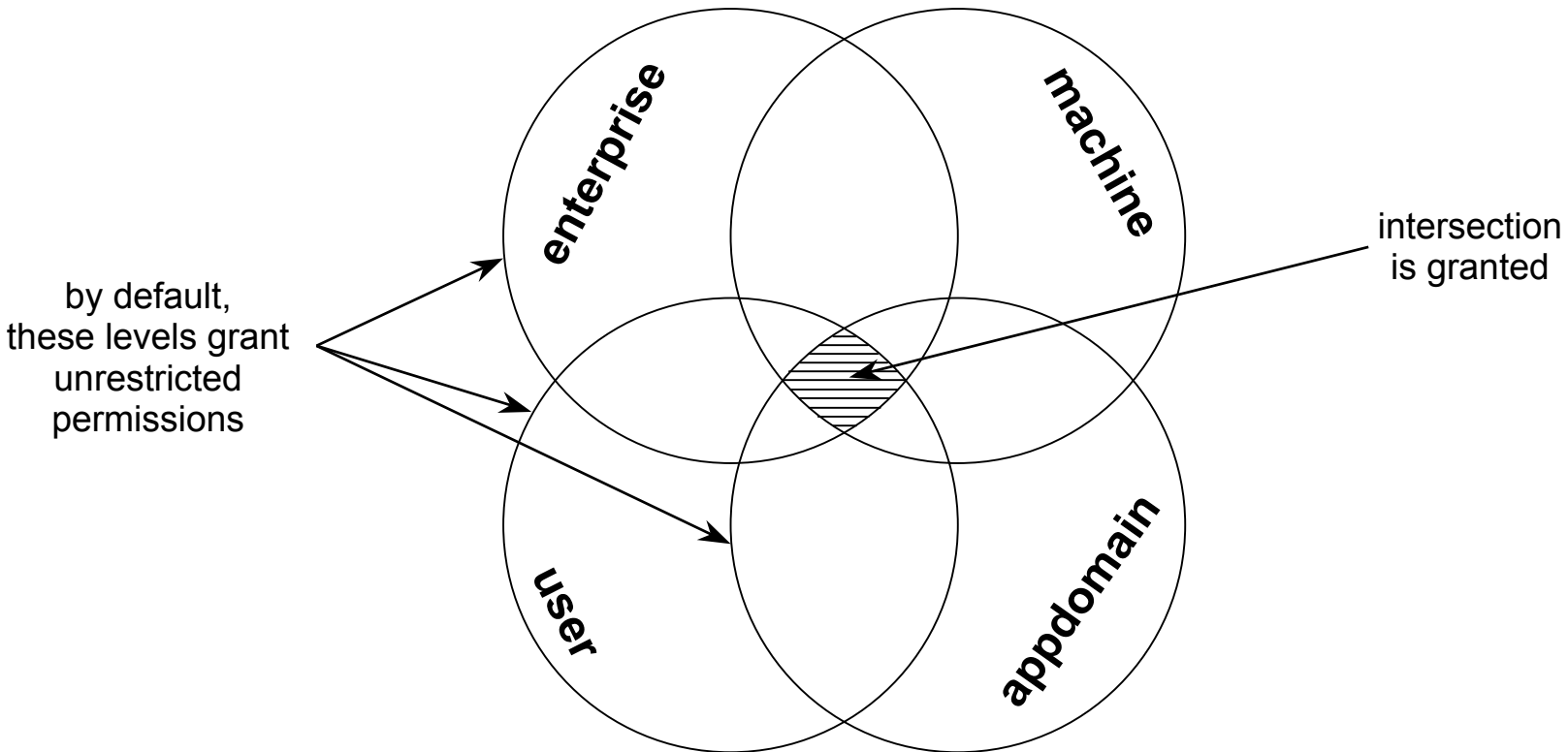
Nothing
Intranet
Same Site Access
FullTrust

= FullTrust



The four policy levels

- Policy comes from four sources
- All must agree before a permission is granted



Policy wrap up

- **Loader discovers evidence**
- **Evidence is fed as input into policy**
- **Each of the four policy levels are evaluated, one at a time**
 - union of permission grants for each matching code group
- **Intersect the permissions granted at each level**
- **Result is permissions granted to assembly**



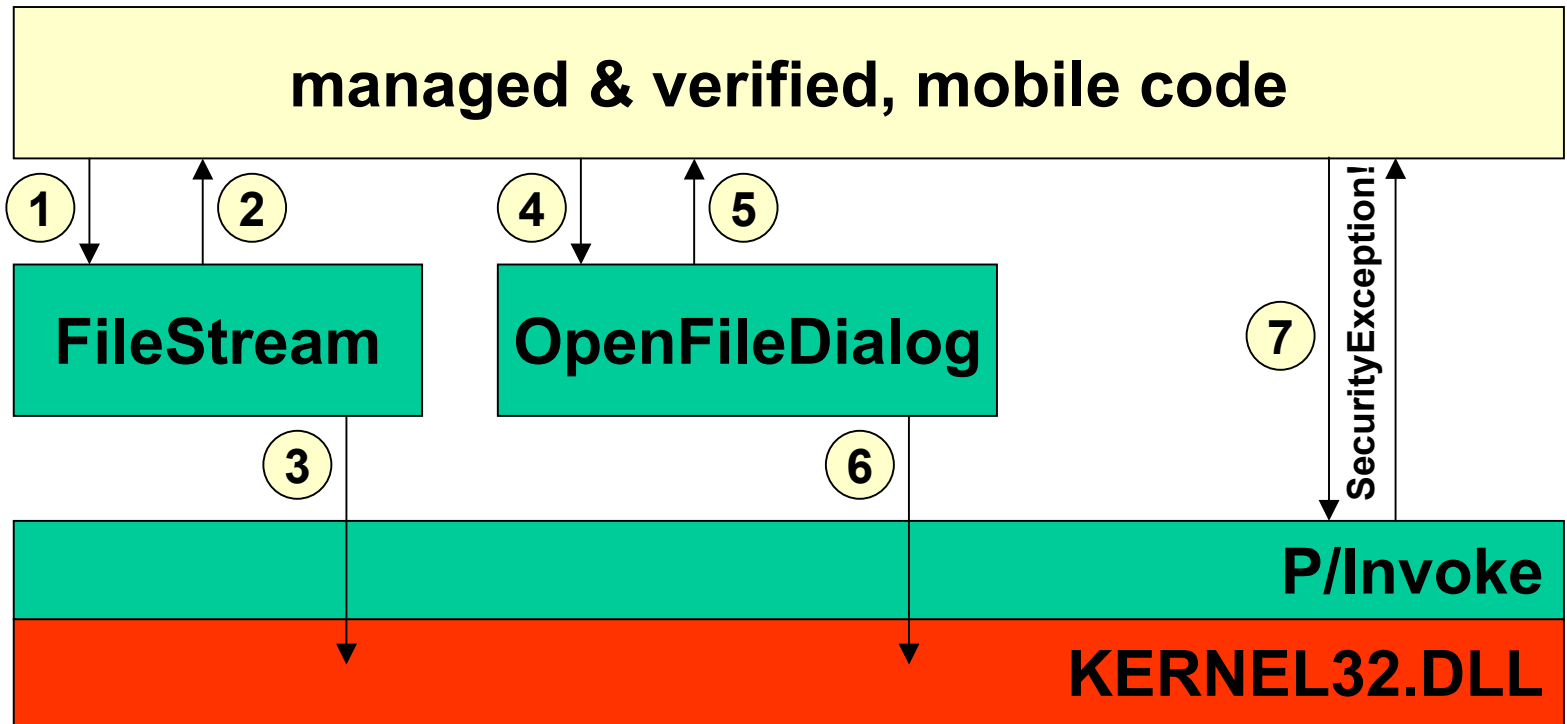
Enforcing security

- **At runtime, the .NET Framework classes demand permissions before performing sensitive operations**
 - Type safe managed code can't get around these checks
 - Mobile code restricted from calling unmanaged code directly
- **Demands may be implemented two ways**
 - Imperative – write the code to call Demand()
 - Declarative – use an attribute to force a demand



Enforcing security

- ② demand FileIOPermission
- ⑤ demand FileDialogPermission
- ⑦ demand SecurityPermission



Example: the file stream constructor

```
// excerpt from mscorlib.dll
public FileStream(string path, FileAccess desiredAccess) {
    FileIOPermissionAccess access = _calcAccess(desiredAccess);

    // describe the action we are going to perform
    CodeAccessPermission perm = new FileIOPermission(access,
                                                    path);

    // demand that the caller can do this
    perm.Demand();

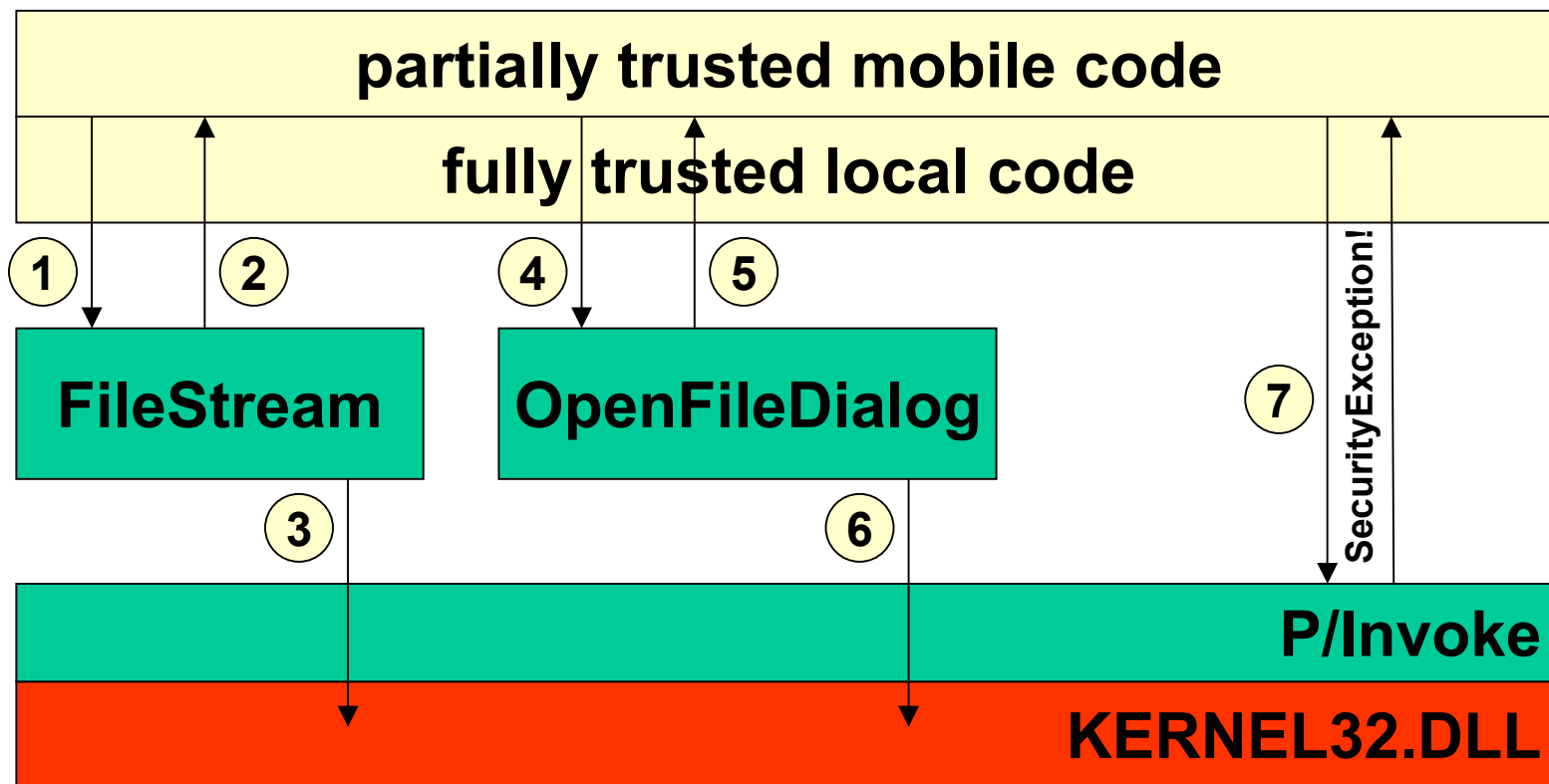
    // call through interop to get a real file handle
    Win32Native.CreateFile(...);
}
```

This is an “imperative”
demand – note we are calling
Demand() programmatically



Security demands actually walk the stack

- Prevents less trusted components from “luring” more trusted components into doing their dirty work



Partially trusted code

- **Partially trusted code can't call strongly named assemblies**
 - not unless the target assembly explicitly allows it
 - local assemblies without strong names cannot be seen by typical mobile code (browser's AppBase is nowhere near your code)
 - assemblies with strong names can be GACked, thus the restriction
- **Why worry?**
- **Isn't CAS strong enough to allow this?**



What if mobile code could use this class?

```
public class HopeThisIsntUsedByEvilCodeBecauseItsNotRobust {
    public string Name;
    public string Password;

    // this entire function consists of
    // horribly bad code you should NEVER EVER use
    public bool IsValidUser() {
        new DbDataPermission(PermissionState.Unrestricted).Assert();
        SqlConnection conn = new SqlConnection(
            "initial catalog=accounts;user id=sa;password=;")
        conn.Open();
        cmd.CommandText = string.Format(
            "select count(*) from users where name='{0}'" +
            "and password='{1}'", Name, Password);
        return ((int)cmd.ExecuteScalar()) > 0;
    }
}
```



How to allow partially trusted code to call you

- **Step 1: review your code for security holes!!!**
- **Step 2: fix the holes!!!**
- **Step 3: goto step 1 until you're really comfortable**
 - Microsoft is still in this loop with many of its assemblies
- **Step 4: apply an attribute to your strongly named assembly**
 - System.Security.AllowPartiallyTrustedCallersAttribute
 - step 4 is much, much easier than steps 1-3, so be vigilant

```
using System.Reflection;
using System.Security;
[assembly: AssemblyKeyFile("publicKey")]
[assembly: AssemblyDelaySign(true)]
[assembly: AllowPartiallyTrustedCallers]
```



Summary

- **CAS is the .NET security model for dealing with mobile code**
- **Evidence is discovered by the loader**
- **Policy takes evidence and turns it into permissions**
- **Permissions say what your code can and cannot do**
- **Permission demands walk the stack to avoid luring attacks**
- **Writing mobile code means understanding CAS**



Deploying Windows Forms Applications

Topics

- **Windows Forms + HTML**
- **Windows Forms as a Better HTML**
- **Application Download**
- **Versioning**
- **Security Zones**
- **Debugging**



Hosting WinForms in HTML

- **Internet Explorer can host Windows Forms controls**
 - <object> tag used to specify type
- **Meant to mimic ActiveX hosting model**
 - Control hosted on the page
 - HTML can program against control
 - Control can fire events into HTML
- **Not quite the same**
 - .NET security model
 - Page must be pulled in from a web server



Example: IE Windows Forms Hosting

```
<!-- Windows Forms control construction -->
<object id="myCtrl" width="100" height="100"
  classid="myCtrl.dll#myCtrl.MyControl">
  <param name="somethingPrefix" value="hi!" />
</object>

<!-- Calling a Windows Forms control method -->
<input type="button" value="Say Something"
  onclick="myCtrl.saySomething('something') "/>

<!-- Handling a Windows Forms control event -->
<script for="myCtrl" event="SomeEvent(arg)">
  alert("Thanks for the " + arg);
</script>
```



Firing Events into IE

- **Firing events requires extra effort**
 - Need to expose events via a COM interface
 - Need to have Full Trust
 - Need to assert permissions before firing event



Example: Firing Events into IE

```
using System.Security.Permissions;
using System.Runtime.InteropServices;

public delegate void SomeCallback(int arg);

// Special COM interface for events
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
public interface IMyEvents {
    // Must match delegate signature
    [DispId(1)] void SomeEvent(int arg);
}

...
```



Example: Firing Events into IE (2)

```
...

// Declaring ourselves as source of these events
[ComSourceInterfaces(typeof(IMyEvents))]
public class MyControl : UserControl {
    ...
    public event SomeCallback SomeEvent;
    void label1_Click(object sender, EventArgs e) {
        // Let hosting code call into unmanaged IE
        SecurityPermissionFlag flag =
            SecurityPermissionFlag.UnmanagedCode;
        SecurityPermission perm = new SecurityPermission(flag);
        perm.Assert();
        if( SomeEvent != null ) SomeEvent(42);
    }
}
```



Controls and the Web Page

- **Downloadable browser controls w/ full permissions unlikely**
 - Destroys “zero touch deployment” benefits of HTML
- **Without events, hosting a Windows Forms control works fine**
 - OK if the control doesn't need to get the page to do things
- **Events allow HTML to act as “glue” for multiple controls**
 - Necessary for controls to interact with each other
- **One control can be entire page...**
 - But what does IE really offer at that point?



“Smart Client”

- **A smart client is**
 - A .NET WinForms application
 - Deployed over the web
 - Lives in a secure sandbox
 - May use web services to talk to the server



Smart Client vs. HTML

- **Smart client pros**
 - Full-featured controls
 - Keyboard handling
 - “Stateful” a.k.a. “no Back button”
 - Familiar interface for users
 - Deployed just like an HTML application
- **Uses a different runtime**
 - i.e. .NET vs. Internet Explorer



Application Download with HTTP

- **Raw HTTP is the simplest approach**
 - Application can be launched with just a URL
 - No special support required on server
 - .NET framework has smart version-aware caching



HTTP Download

- Applications are downloaded via URLs
 - Normal HTTP protocol applies

```
GET /foo.exe HTTP/1.1
```

```
Accept: */*
```

```
Accept-Language: en-us
```

```
Accept-Encoding: gzip, deflate
```

```
User-Agent: Moz
```

```
(compatible; MS
```

```
5.1; Q312461; .
```

```
Host: localhost
```

```
Connection: Kee
```

```
HTTP/1.1 200 OK
```

```
Server: Microsoft-IIS/5.1
```

```
Date: Fri, 01 Feb 2002 02:11:29 GMT
```

```
Content-Type: application/octet-stream
```

```
Accept-Ranges: bytes
```

```
Last-Modified: Fri, 01 Feb 2002 01:41:16 GMT
```

```
ETag: "50aae089c1aac11:916"
```

```
Content-Length: 45056
```

```
<<stream of bytes from foo.exe>>
```

Caching

- **Downloaded assemblies are kept in the internet cache**
 - Only downloaded if server has newer bits

```
GET /foo.exe HTTP/1.1
```

```
Accept: */*
```

```
Accept-Language: en-us
```

```
Accept-Encoding: gzip, deflate
```

```
If-Modified-Since: Fri, 01 Feb 2002 01:41:16 GMT
```

```
If-None-Match: "50aae089c1aac11:916"
```

```
User-Agent: Mozilla/4.0 ...
```

```
Host: localhost
```

```
Connection: Keep-Alive
```

```
HTTP/1.1 304 Not Modified
```

```
Server: Microsoft-IIS/5.1
```

```
Date: Fri, 01 Feb 2002 02:42:03 GMT
```

```
ETag: "a0fa92bc8aac11:916"
```

```
Content-Length: 0
```



Versioning

- **.NET has built in support for versioning**
 - Only enabled if versioned assembly is signed
- **Initial assembly not versioned**
 - URL has no version info
 - Uses If-Modified-Since to get “latest version”
- **Referenced assemblies are versioned**
 - No HTTP request is made if they’re cached



Related Assemblies

- **Related assemblies are pulled in as needed**
 - Can be implicit via /r at compile-time
 - Can be explicit via Load/LoadFrom at run-time

```
void AddDynamicControl() {  
    Assembly assem =  
        Assembly.LoadFrom("http://localhost/mycontrol.dll");  
    Control ctrl = (Control)assem.CreateInstance("MyControl");  
    ctrl.Location = new Point(10, 10);  
    ctrl.Size = new Size(100, 100);  
    Controls.Add(ctrl);  
}
```



Optimizing the Resource Manager

- **RM looks in the “appbase”**
 - i.e. the assembly’s directory
 - For web-deployed assemblies, the appbase is the base URL
 - Extra round-trips take time over the web
- **Tell the RM the culture of the assembly**
 - [assembly: NeutralResourcesLanguageAttribute("en-US")]
- **Empty files work for other cultures**
 - Or don’t let Designer generate your resource management code



Debugging URL-launched Applications

- **Sometimes things go wrong...**
 - Debugging needs to happen in semi-trust context
 - Nothing built into VS.NET to make that happen
- **Use IeExec.exe as start application**
 - IeExec used to host downloaded assemblies



ieExec.exe

```
Usage: ieexec.exe url flags [securityZone] [domainId]
url           Assembly to launch, e.g. http://foo/foo.exe
flags        Flags to control execution. Values that can be
             added together are:
             0:   no flags
             1:   create evidence for the zone
             2:   create evidence for the site
securityZone If evidenceFlags != 0, sets the security zone.
             Values can be {0, 1, 2, 3} for
             {MyComputer, Intranet, Trusted, Internet}
domainId     If evidenceFlags != 0, unused hex-encoded
             bytes. Use 00.
```



Debugging in VS.NET

Debuggers	
Enable ASP Debugging	False
Enable ASP.NET Debugging	False
Enable Unmanaged Debugging	False
Enable SQL Debugging	False
Start Action	
Debug Mode	Program
Start Application	NET\Framework\v1.0.3705\IExec.exe ...
Start URL	
Start Page	
Start Options	
Command Line Arguments	http://foo/foo.exe 0
Working Directory	
Always use Internet Explorer	True
Enable remote debugging	False

Limitations of URL-launched Applications

- **Must have connectivity to web server**
 - (Can work out of .NET download cache when disconnected, but this only works in Work Offline selected in IE)
- **Updates only detected at application launch time**
 - User must quit and restart to pick up updates
 - No built-in mechanism for notifying user that this is necessary
- **User has no choice about when to retrieve updates**
 - Can get expensive on GPRS...
- **More sophisticated hybrid approaches possible**
 - E.g., AppUpdater



Hybrid Deployment

- **Application installed locally in normal way**
 - E.g. MSI file, or xcopy
- **Periodically checks for updates (usually via HTTP)**
 - May use BITS to download updates when available
- **User notified when new version ready**
- **Advantages:**
 - Can launch application when network not available
 - Potential for more flexible download options (e.g., don't download when connected via GPRS...)
- **Disadvantages:**
 - No built-in infrastructure to support this (yet)
 - Security policy no longer applied automatically
 - Initial local installation step required (no longer 'zero-touch')



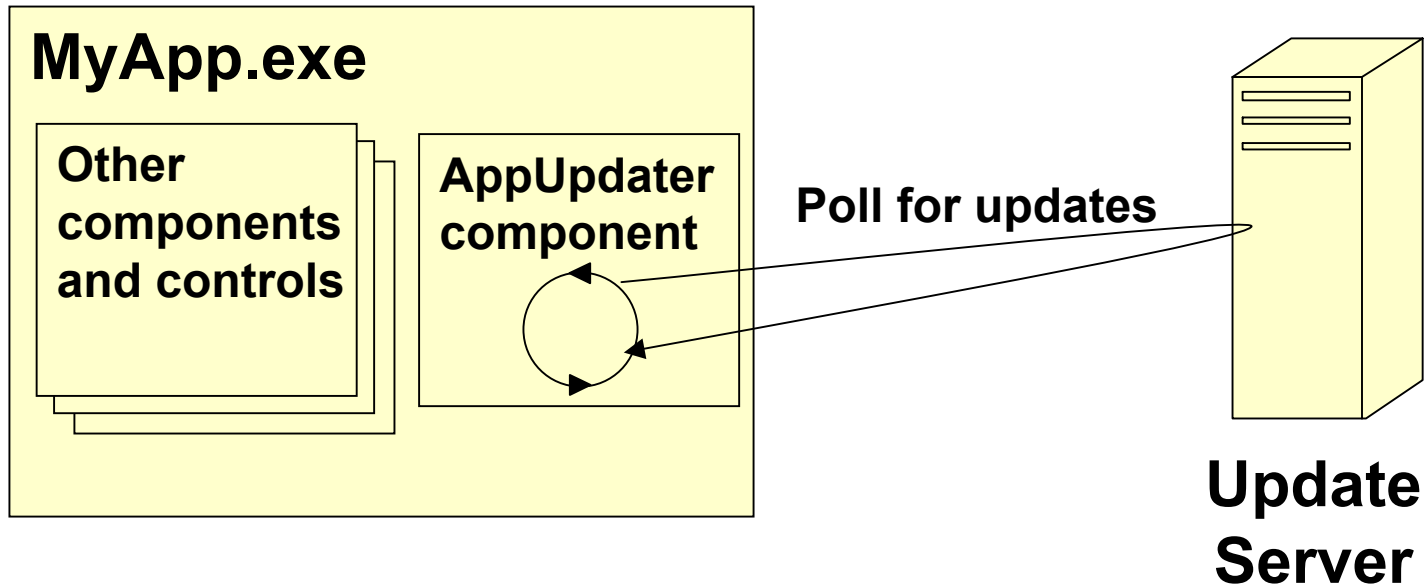
AppUpdater

- **AppUpdater illustrates the hybrid technique:**
 - <http://www.windowsforms.net/articles/appupdater.aspx>
- **This is SAMPLE code – not a supported product**
- **Update mechanism has a local security issue:**
 - Updates downloaded to subdirectory of application directory
 - Instructions suggest installing app in c:\Program Files\
– Admin privileges therefore required to download updates!



How AppUpdater Works

- **AppUpdate component regularly polls for updates**
 - Can use web service
 - Can download an XML info file from web server



AppUpdater Download

- **Once an update is available download commences**
- **AppUpdater uses HTTP-DAV**
 - Fires up a download thread
 - Uses HTTP-DAV to discover what files are available
 - Downloads all necessary files
- **BITS (Background Intelligent Transfer Service) would be best**
 - Offers background transfer without hogging network bandwidth
 - Transfers continue when application exits
 - Robust – will retry after disconnects and even after reboots
 - Not available on Win9x or WinME
 - Consequently not supported by AppUpdater



AppUpdater Applying Updates

- **Once updates are downloaded, they must be applied**
- **Cannot simply copy into application directory**
 - If the application is already running, this will fail
- **AppUpdater copies updates into a new subdirectory**
- **Relies on 'AppStart.exe' bootstrap program**
 - When launched, AppStart looks for latest downloaded version
 - Creates new child AppDomain and runs program in there
 - Can be configured to run program in separate process
 - Old versions deleted when user exits and restarts
- **Could also have used shadow copies**
 - Same technique ASP.NET uses – runs copies of binaries
 - AppUpdater avoids this as it is marginally less efficient



Security in Hybrid Deployment

- **AppUpdater-style deployment downloads code to local disk**
 - CLR therefore considers code to be in My Computer zone
 - Code consequently granted Full Trust by default
 - Should therefore be careful about what you download
- **Several options exist for dealing with this**
 - HTTPS – trust items because they came from the right place
 - Strong Naming – trust signed items (AppUpdater does this)
 - Sandbox – create your own sandbox



DIY Security Sandbox

- Can create AppDomain with restricted security
- Pass Evidence to AppDomain.CreateDomain
 - E.g., run local assembly as if launched from Intranet zone:

```
using System;
using System.Security;
using System.Security.Policy;
...
void RunRestricted(string exePath) {
    Evidence ev = new Evidence();
    ev.AddHost(new Zone(SecurityZone.Intranet));
    AppDomain childApp = AppDomain.CreateDomain("Host", ev);
    childApp.ExecuteAssembly(exePath);
}
```



Summary

- **Smart apps offer Deployment model of HTML**
- **UI richness of Windows**
- **Ease of development of RAD**
- **Stateful programming model**
- **.NET smart clients combine “the best of all worlds”**
 - Disclaimer: requires .NET runtime...
- **AppUpdater-style deployment offers greater flexibility**
 - No longer ‘no touch’ – at least ‘one-touch’

