WICKED CODE

3 Important Tips For Silverlight Development

Jeff Prosise

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY
Browse the Code Online

Contents

As I write this, Silverlight 2 is hot off the presses and developers are getting a first look at what many believe represents the future of Web programming. Whether you're a Silverlight proponent or find more allure in competing technologies such as Adobe Flex, it's exciting to see alternatives to HTML, JavaScript, and AJAX emerge and gain mindshare for creating Web applications. And with Microsoft already hard at work on Silverlight 3, the future has never seemed brighter.

As is true of any platform, the road to becoming a Silverlight developer is not without a few potholes. Did you know, for example, that many calls to XamlReader.Load that test fine on PCs in the United States will fail on PCs in other countries? Did you realize that Silverlight's rendering engine is intimately tied to the UI thread and that this fact can profoundly impact the structure of your code? Did you know that you can reduce the size of your XAP files by dynamically loading assemblies, but that doing so without losing the benefit of strong typing requires knowledge of CLR internals? If this intrigues you, read on. I have some tips and tricks to share that will make life with Silverlight a little less bumpy—and make you a better and more informed Silverlight programmer, too.

For more on packaging Silverlight content for faster delivery, please see the January 2009 installment of Cutting Edge.

On-Demand Assembly Loading

One of the hallmarks of a well-designed Silverlight application is a small XAP file, formally known as an application package. XAP files all too often swell to unmanageable sizes as the result of embedded resources (especially images) and assembly references. The larger the XAP file, the longer it takes to download, and if it grows too large, Silverlight might be unable to load it.

Even large applications can be packaged in small XAP files if you're careful to factor the resources and assemblies that the application uses and keep those that can be delay-loaded or downloaded on demand on the Web server. You can use WebClient or other classes in Silverlight's networking stack to download additional resources and assemblies once the application package has been downloaded. It's generally preferable to get the application's UI up and running quickly and then launch asynchronous network requests for the additional assets you need than it is to post a 100MB XAP file and force the user to spend five minutes waiting for a progress indicator to reach 100%.

On-demand resource loading in Silverlight tends to be simple and straightforward. The code snippet in **Figure 1**, for example, downloads a JPEG deployed at the site of origin and displays it by assigning the downloaded bits to a XAML image named MyImage.

Figure 1 Downloading an Image from the Site of Origin

Copy Code

```
WebClient wc = new WebClient();
wc.OpenReadCompleted +=
    new OpenReadCompletedEventHandler(wc_OpenReadCompleted);
wc.OpenReadAsync(new Uri("JetCat.jpg", UriKind.Relative));
  ...
void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    if (e.Error == null)
    {
```

```
        BitmapImage bi = new BitmapImage();
        bi.SetSource(e.Result);
        MyImage.Source = bi;
    }
}
```

On-demand assembly loading tends to be more difficult, however. At first glance it seems easy: use WebClient to download the assembly and AssemblyPart.Load to load it into the appdomain. The problem is that Silverlight's JIT compiler can get in the way, leading many developers to believe that it's impossible to download assemblies on demand and enjoy the benefits of strong typing, too. In reality, however, you can do both. But you need to know what you're doing and having a basic understanding of how assembly loading works in the CLR.

To demonstrate, consider the following code:

Copy Code

```
private void CreateCalendarButton_Click(object sender, RoutedEventArgs e)
{
    Calendar cal = new Calendar();
    cal.Width = 300.0;
    cal.Height = 200.0;
    cal.SelectedDatesChanged += new
        EventHandler<SelectionChangedEventArgs>(cal_SelectedDatesChanged);
    LayoutRoot.Children.RemoveAt(0);
    LayoutRoot.Children.Add(cal);
}
```

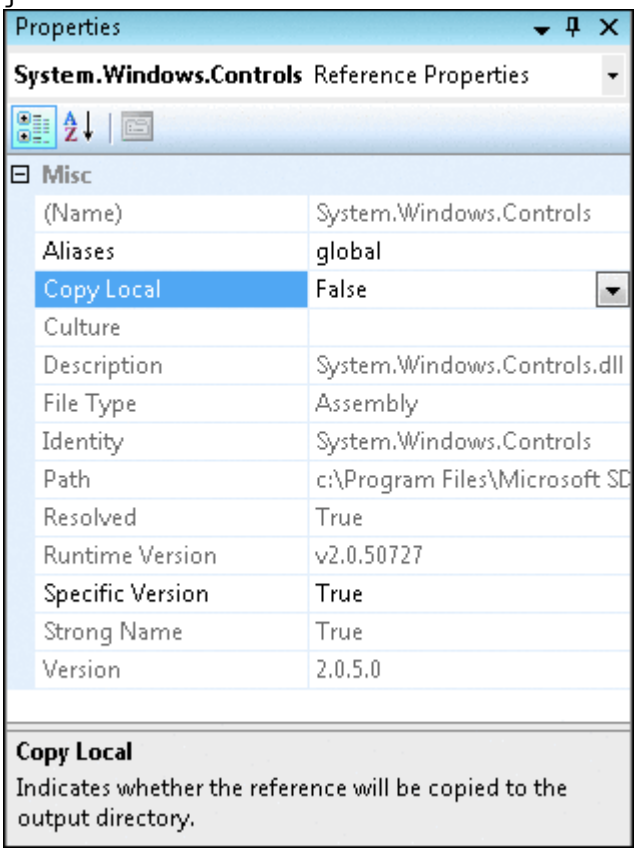

Figure 2 **Keeping an Assembly out of the XAP**

It's a button click handler that dynamically creates a Calendar control and adds it to the XAML scene. (It also deletes the button that fired the event, which is assumed to be the 0th item in LayoutRoot's Children collection.) Because Calendar is implemented in the System.Windows.Controls.dll, which isn't embedded in the Silverlight plug-in but is instead part of the extended BCL, this code works just fine as long as you add a

reference to System.Windows.Controls.dll to your project. The reference causes System.Windows.Controls.dll to be included in the XAP file and automatically loaded into the appdomain. Now suppose you want to be clever and only load System.Windows.Controls.dll if it's needed—that is, if the user clicks the button. So you add a reference to System.Windows.Controls.dll to the project to satisfy the compiler (otherwise, the compiler won't compile a reference to Calendar because the compiler has no idea what the Calendar type is) and, in the Visual Studio Properties window, you set System.Windows.Controls.dll's Copy Local property to false to prevent it from being embedded in the XAP file (as I did in **Figure 2**).

Next, you deploy a copy of System.Windows.Controls.dll alongside the XAP file in the application's ClientBin folder on the server. Finally, you restructure your code as shown in **Figure 3**. The button click handler now downloads System.Windows.Controls.dll from the Web server, loads it into the appdomain with AssemblyPart.Load, and instantiates a Calendar control.

Figure 3 On-Demand Assembly Loading that Doesn't Work

Copy Code

```
private void CreateCalendarButton_Click(object sender, RoutedEventArgs e)
{
    WebClient wc = new WebClient();
    wc.OpenReadCompleted +=
        new OpenReadCompletedEventHandler(wc_OpenReadCompleted);
    wc.OpenReadAsync(new Uri("System.Windows.Controls.dll",
        UriKind.Relative));
}

void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    if (e.Error == null)
    {
        // Load the downloaded assembly
        AssemblyPart part = new AssemblyPart();
        part.Load(e.Result);

        // Create a Calendar control
        Calendar cal = new Calendar();
        cal.Width = 300.0;
        cal.Height = 200.0;
        cal.SelectedDatesChanged += new
           EventHandler<SelectionChangedEventArgs>(cal_SelectedDatesChanged);
        LayoutRoot.Children.RemoveAt(0);
        LayoutRoot.Children.Add(cal);
    }
}
```

It looks reasonable and the code compiles just fine, but at run time, the click handler generates an exception like the one in **Figure 4**. Code that compiles is good. Code that throws exceptions is not. So what gives? The error message seems to indicate that the CLR is trying to load System.Windows.Controls.dll, but it shouldn't need to since you're loading it programmatically.

This is a great example of a case where knowledge of CLR internals can make you a better Silverlight programmer. The problem here is that when the JIT compiler compiles your wc_OpenReadCompleted method, it scans the method, sees that it references a type named Calendar, and attempts to load System.Windows.Controls.dll so that the reference can be resolved.

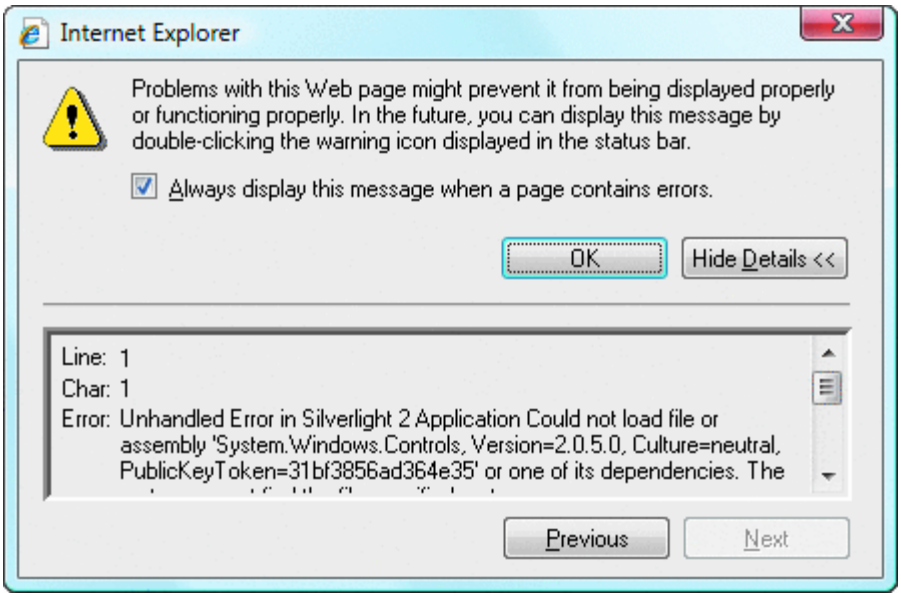

Figure 4 **Oops!**

Unfortunately, this happens before the method is even executed, thus you don't get the chance to call AssemblyPart.Load. It's a classic chicken-and-egg problem. You need to call AssemblyPart.Load to load the assembly, but before you can call it, the JIT compiler intervenes and attempts to load it for you. The attempt fails because System.Windows.Controls.dll isn't in the application package.

This is the point at which many programmers throw up their hands and either conclude that on-demand assembly loading doesn't work in Silverlight or resort to reflection to instantiate the Calendar type:

Copy Code

```
AssemblyPart part = new AssemblyPart();
Assembly a = part.Load(e.Result);
Object cal = (Object)a.CreateInstance("Calendar");
```

This approach works, but it's clumsy. You can't cast the reference returned by Assembly.CreateInstance to a Calendar because doing so would cause the JIT compiler to attempt to load the assembly before the method executed. And if you can't cast to Calendar, then the control's methods, properties, and events have to be accessed through reflection, too. The code quickly grows so unwieldy that it's tempting to just give in and embed System.Windows.Controls.dll in the application package and live with the increased XAP size.

The good news is that you can combine dynamic assembly loading and strong typing. Simply restructure your code along the lines of **Figure 5**. Observe that wc_OpenReadCompleted no longer references the Calendar type; all references have been moved to a separate method named CreateCalendar. Furthermore, CreateCalendar is attributed in such a way that the JIT compiler will not attempt to inline the method. (If inlining were to occur, you'd be right back where you started because wc_OpenReadCompleted would contain an implicit reference to the Calendar type.) Now the JIT compiler won't check to see if System.Windows.Controls.dll has been loaded until CreateCalendar is called, and by that time, you've already loaded it into the appdomain.

Figure 5 On-Demand Assembly Loading that Works

Copy Code

```
private void CreateCalendarButton_Click(object sender, RoutedEventArgs e)
{
    WebClient wc = new WebClient();
    wc.OpenReadCompleted +=
        new OpenReadCompletedEventHandler(wc_OpenReadCompleted);
    wc.OpenReadAsync(new Uri("System.Windows.Controls.dll",
        UriKind.Relative));
}

void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
```

```
{
    if (e.Error == null)
    {
        // Load the downloaded assembly
        AssemblyPart part = new AssemblyPart();
        part.Load(e.Result);

        // Create a Calendar control
        CreateCalendar();
    }
}

[MethodImpl(MethodImplOptions.NoInlining)]
private void CreateCalendar()
{
    Calendar cal = new Calendar();
    cal.Width = 300.0;
    cal.Height = 200.0;
    cal.SelectedDatesChanged += new
        EventHandler<SelectionChangedEventArgs>(cal_SelectedDatesChanged);
    LayoutRoot.Children.RemoveAt(0);
    LayoutRoot.Children.Add(cal);
}
```

Insights: Rendering and the UI Thread

Notice that Jeff said that one aspect of Silverlight that doesn't get a lot of attention is the fact that all rendering in Silverlight is done on the application's UI thread, and if you hog the UI thread, you prevent any rendering from happening. Since WPF provides a rendering thread, it is probably surprising that Silverlight does not. You may be interested to know why.

The decision came down to a tradeoff between system overhead and decoupling framerates. With Silverlight, we went with a lighter weight on-thread approach and did not isolate your application code from the rendering system. That means you can do more in your animation (like have layout-based animation or custom code running) and there is minimal latency and overhead getting to the rendering system. The down side is if you do too much, you can interfere with operations such as video playback.

That said, the Silverlight rendering system will take advantage of multi-core processing and use many threads to speed up rendering for it. So, rendering is rarely "on thread," but it is synchronized with your app to avoid synchronization plus copies of data.

—Ashraf Michail, Principal Architect, Silverlight

Incidentally, if this were Windows Presentation Foundation (WPF) rather than Silverlight, you could resolve the problem in a more elegant manner by registering a handler for AppDomain.AssemblyResolve events and loading System.Windows.Controls.dll there. AppDomain.AssemblyResolve exists in Silverlight, but it's attributed SecurityCritical, which means user code can't register handlers for it.

Figure 5 assumes that you included in your project a reference to System.Windows.Controls.dll, but that you set Copy Local to false (see **Figure 2**) and deployed the assembly in the ClientBin folder. To prove that it works, download the OnDemandAssemblyDemo application that accompanies this column and click the button labeled Create Calendar Control. A Calendar control appears in place of the button. Significantly, OnDemandAssemblyDemo.xap does not contain a copy of System.Windows.Controls.dll, which you can easily verify by opening the XAP file with WinZip. Magic! This will be a great ice-breaker for your next Silverlight party.

Just-in-Time Rendering

One aspect of Silverlight that doesn't get a lot of press is the fact that all rendering in Silverlight is done on the application's UI thread, and if you hog the UI thread, you prevent any rendering from being done. That means you want to avoid long-running loops on the UI thread if you're modifying a XAML scene in that loop or if any animations are taking place at the same time.

It sounds simple—avoiding long-running loops on the UI thread—but in practice, it can have a profound impact on the code that you write. Consider the app called OpenFileDialogDemo, pictured in **Figure 6**. It demonstrates how to use Silverlight's OpenFileDialog class to allow the user to browse his or her hard disk for image files and then load the images into XAML image objects. Run the app, click the Open button at the top of the page, select several image files (the bigger the better), and click the OpenFileDialog's Open button.

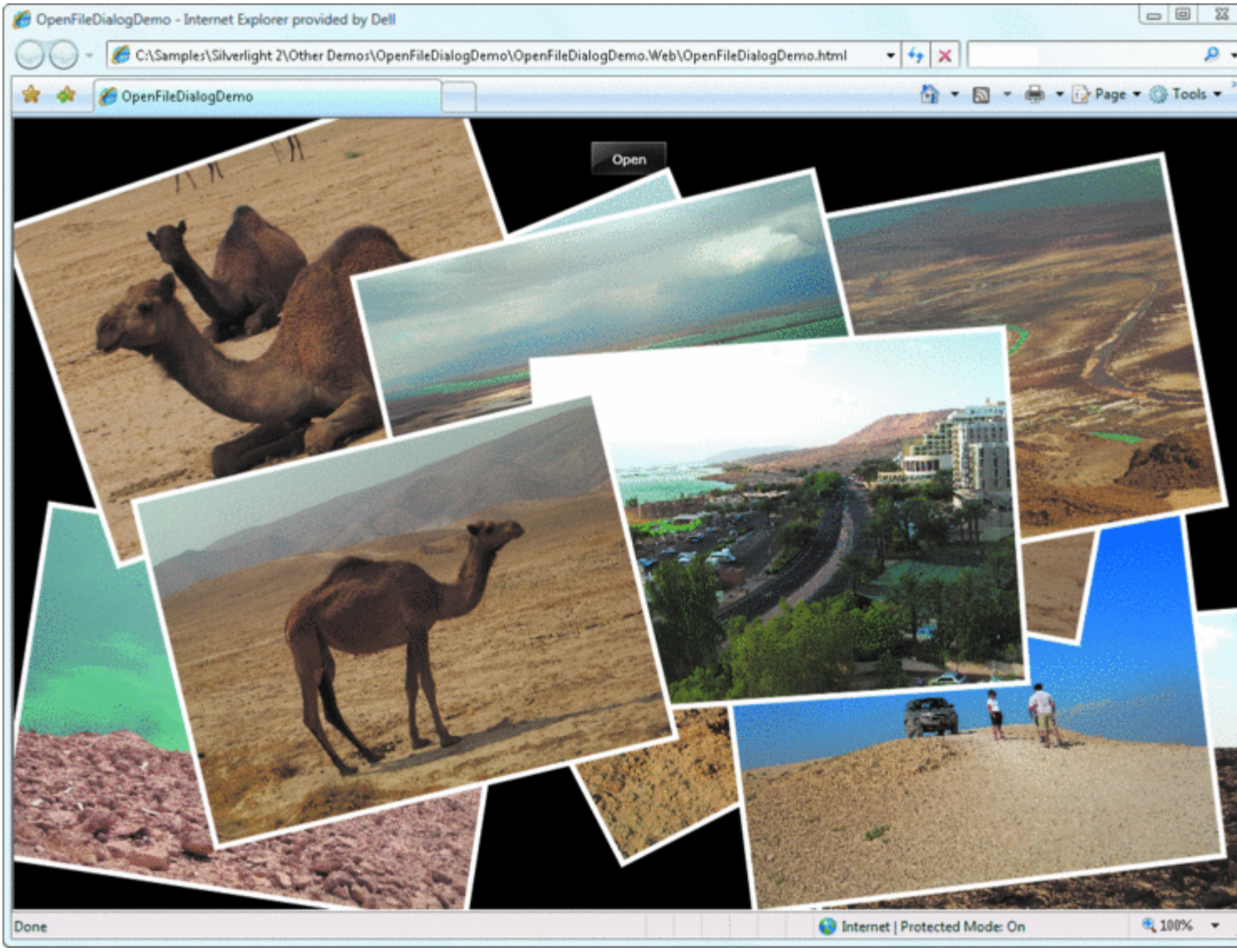

Figure 6 **OpenFileDialogDemo in Action**

You'll see that one by one, the images you selected pop into the scene using objects created dynamically with XamlReader.Load and assume random positions on the page. Once the images are displayed, you can click them to make them come to the front and even use the mouse to drag them around the page.

Despite its apparent simplicity, OpenFileDialogDemo offers a practical lesson in being judicious with the UI thread. When I originally wrote the code to display the OpenFileDialog and load the image files, I structured it something like the snippet in **Figure 7**. Once the user has dismissed the dialog, a simple foreach loop iterates through the selected files and loads them one by one.

Figure 7 Simple Approach to Loading Image Files

Copy Code

```
OpenFileDialog ofd = new OpenFileDialog();
ofd.Filter = "JPEG Files (*.jpg;*.jpeg)|*.jpg;*.jpeg|" +
```

```
    "PNG Files (*.png)|*.png|All Files (*.*)|*.*";
ofd.FilterIndex = 1;
ofd.Multiselect = true;

if ((bool)ofd.ShowDialog())
{
    foreach (FileInfo fi in ofd.Files)
    {
        using (Stream stream = fi.OpenRead())
        {
            BitmapImage bi = new BitmapImage();
            bi.SetSource(stream);
            GetNextImage().Source = bi;
        }
    }
}
```

Unfortunately, none of the images appeared on the screen until all of them were loaded. The delay wasn't a big deal if the user selected one or two image files, but it was intolerable if 40 or 50 files were selected. In short, the app didn't meet the minimum requirements I set for it, because I wanted the images to "pop" onto the screen as they were loaded. Get it? Pop! Pop! Pop!

The problem, of course, was that the foreach loop runs on the UI thread, and while the loop was running, Silverlight couldn't render the images as they were added to the scene—which meant it was time to step back, take a breath, and restructure the code to make it able to render the images in a timely manner. **Figure 8** shows one solution to the problem. The modified foreach loop does nothing more than add FileInfo objects to a System.Collections.Generic.Queue. This enables the loop to run quickly and hand control back to Silverlight so it can get down to the business of rendering. Perhaps the most interesting aspect of the restructured code is how it dequeues and processes the FileInfo objects in response to CompositionTarget.Rendering events.

Figure 8 A Better Way to Load Image Files

Copy Code

```
private Queue<FileInfo> _files = new Queue<FileInfo>();
 ...
public Page()
{
    InitializeComponent();

    // Register a handler for Rendering events
    CompositionTarget.Rendering +=
        new EventHandler(CompositionTarget_Rendering);
}
 ...
OpenFileDialog ofd = new OpenFileDialog();
ofd.Filter = "JPEG Files (*.jpg;*.jpeg)|*.jpg;*.jpeg|" +
    "PNG Files (*.png)|*.png|All Files (*.*)|*.*";
ofd.FilterIndex = 1;
ofd.Multiselect = true;

if ((bool)ofd.ShowDialog())
{
    // Reset the queue
    _files.Clear();

    // Place each FileInfo in a queue
    foreach (FileInfo fi in ofd.Files)
    {
        _files.Enqueue(fi);
```

```
    }
}
 ...
private void CompositionTarget_Rendering(Object sender, EventArgs e)
{
    if (_files.Count != 0)
    {
        FileInfo fi = _files.Dequeue();
        using (Stream stream = fi.OpenRead())
        {
            BitmapImage bi = new BitmapImage();
            bi.SetSource(stream);
            GetNextImage().Source = bi;
        }
    }
}
```

CompositionTarget.Rendering is a per-frame rendering callback traditionally used to implement game loops. It was borrowed from WPF and showed up late in the Silverlight 2 development cycle. The event is fired each time Silverlight is ready to re-render the scene.

OpenFileDialogDemo registers a handler for CompositionTarget.Rendering events (CompositionTarget_Rendering) and dequeues one FileInfo object, converting it into a XAML image, each time the handler is called. The result? The images pop onto the screen as they're loaded because Silverlight now has the opportunity to update the scene following the addition of each new image. This is how the foreach loop in the final version of OpenFileDialogDemo is structured, and it's why when you ran it, you saw images appear on the screen one by one instead of all at once.

You want to be careful not to overuse CompositionTarget.Rendering. If OpenFileDialogDemo had animations running as it added images to the scene, the animations would likely stutter because each frame would be delayed by the amount of time required to load the image bits and assign them to an image object. But when you need it, you need it badly, and OpenFileDialogDemo is a good example of an acceptable use of CompositionTarget.Rendering—indeed, here the goal would be difficult to accomplish otherwise.

Avoiding Regional Dependencies

A final tip regards using XamlReader.Load to create XAML objects dynamically. Can you spot what's wrong with this code?

Copy Code

```
Rectangle rect = (Rectangle)XamlReader.Load(
    String.Format(
        "<Rectangle xmlns=\"http://schemas.microsoft.com/client/2007\" " +
        "Width=\"{0}\" Height=\"{1}\" Stroke=\"Black\" Fill=\"Yellow\" />",
        100.5, 100.0
    )
);
```

If you recognized that this code will work on most PCs in the United States but will fail on most PCs in Europe and in other parts of the world, give your self a pat on the back. To demonstrate, first configure your operating system to display numbers, currencies, dates, and times in U.S. format if it isn't configured that way already. (In Vista, go to the Format tab of the Regional and Language Options dialog accessible through the Control Panel.) Execute the call to XamlReader.Load and verify that the call executes successfully. Now change the regional format to French and execute the call again. This time XamlReader.Load throws an exception:"Invalid attribute value 100,5 for property Width" (**Figure 9**). The problem is that decimal numbers such as 100.5 are written 100,5 (notice the comma in place of the decimal point) in many countries. And since String.Format honors regional settings on the host PC, the decimal 100.5 becomes "100,5". Unfortunately, XamlReader.Load doesn't know what to make of "100,5", so it throws an exception.

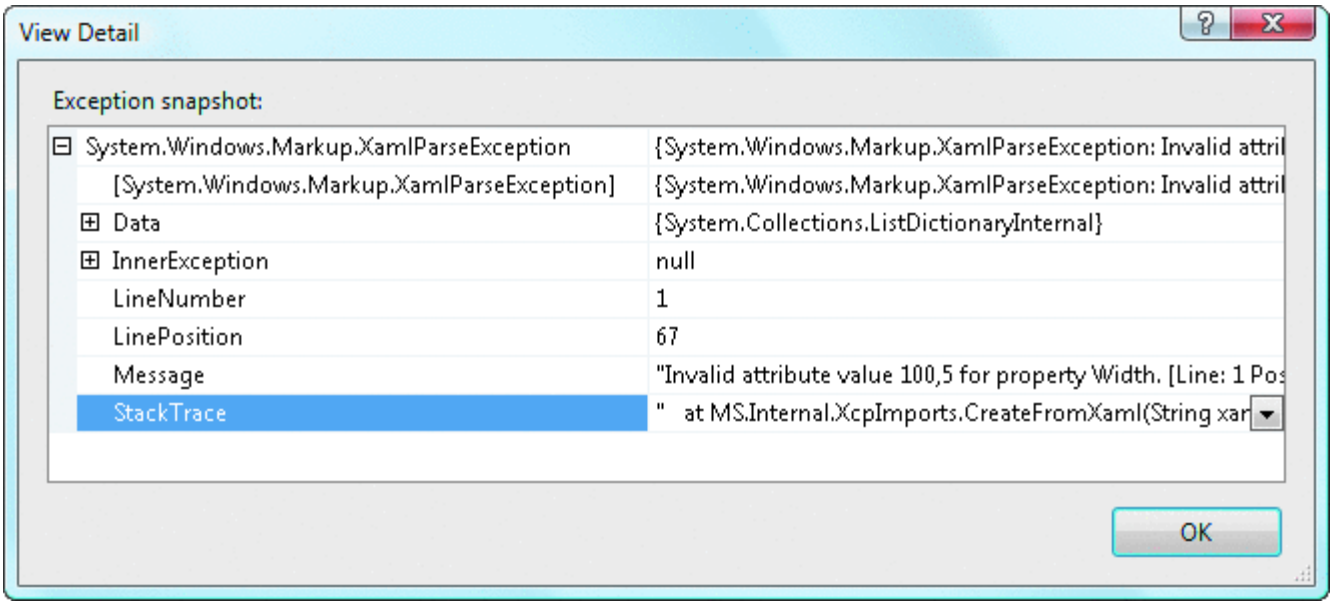

Figure 9 **Exception Thrown by XamlReader.Load**

The following code shows the correct way to call XamlReader.Load so that it works on any PC that runs Silverlight:

Copy Code

```
Rectangle rect = (Rectangle)XamlReader.Load(
    String.Format(
        CultureInfo.InvariantCulture,
        "<Rectangle xmlns=\"http://schemas.microsoft.com/client/2007\" " +
        "Width=\"{0}\" Height=\"{1}\" Stroke=\"Black\" Fill=\"Yellow\" />",
        100.5, 100.0
    )
);
```

The first parameter passed to String.Format is a CultureInfo object referencing the invariant culture. XamlReader.Load expects culture-invariant strings, so using CultureInfo.InvariantCulture ensures that String.Format generates properly formatted decimal values (as well as properly formatted dates and times if you're using those). Not coincidentally, the OpenFileDialogDemo application in the previous section uses this technique to make sure that its own calls to XamlReader.Load work regardless of regional settings.

If the strings you pass to XamlReader.Load include decimal values generated by String.Format, always use CultureInfo.InvariantCulture to get proper formatting. Once your application is running in the wild, it's liable to execute under a variety of regional settings.

Turning a New Page

If you're beginning to use Silverlight and you're an experienced .NET developer, you already know 90% of what you need to know. But Silverlight brings nuances to .NET that you should understand.

Speaking of pages, many readers have asked when I intend to update the Silverlight 1.0 page-turn framework presented in May 2008 for Silverlight 2. Well, porting is complete. You can view a sample app that uses the updated framework at wintellect.com/silverlight/pageturndemo/, and you can download the source code from wintellect.com/Downloads/PageTurnDemo2.zip.

The page-turn framework lives in PageTurn.cs, and the code in Page.xaml.cs illustrates how it all works. The API is similar to the Silverlight 1.0 version (in C# rather than JavaScript), and I made changes to make page turning better. I included a PageTurned event that's fired by the framework each time a page turn is complete so you can update the UI—say, to show the current page number.

Send your questions and comments for Jeff to wicked@microsoft.com.

Jeff Prosise is a contributing editor to *MSDN Magazine* and the author of several books, including Programming Microsoft .NET (2002, Microsoft Press). He's also cofounder of Wintellect (wintellect.com), a

software consulting and education firm that specializes in Microsoft .NET. Contact Jeff at wicked@microsoft.com.