First Look: Microsoft® Silverlight™ 2

Laurence Moroney
# Table of Contents

## 1  Introducing Silverlight 2 ................................................................. 3
  Silverlight and User Experience................................................................. 3
  Silverlight Architecture ........................................................................ 5
  Silverlight and XAML ............................................................................. 7
  Silverlight and the Expression Suite ....................................................... 11
    Silverlight and Expression Design ...................................................... 11
    Silverlight and Expression Blend ....................................................... 14
  Silverlight and Expression Encoder ......................................................... 19
  Summary ................................................................................................. 22

## 2  Using Expression Blend with Silverlight 2 ........................................... 23
  Getting Started with Expression Blend ...................................................... 23
    Creating a Silverlight 2 Application .................................................. 25
  The Expression Blend IDE ........................................................................ 30
    The Tools Pane ...................................................................................... 30
    The Interaction Pane ............................................................................ 31
    The Design Surface ............................................................................ 32
    The Project Pane ................................................................................. 33
    The Properties Pane ........................................................................... 35
  Using Blend to Build Silverlight Applications ......................................... 36
    Layout .................................................................................................. 36
    Placing and Customizing Visual Elements ............................................ 43
    Placing and Customizing Controls ....................................................... 44
    Using Blend to Design Animations ...................................................... 45
  Summary ................................................................................................. 48

## 3  Using Visual Studio with Silverlight 2 ................................................. 49
  Installing the Visual Studio Tools for Silverlight ..................................... 49
  Using Visual Studio to Build a Silverlight Application .......................... 51
    Creating a Silverlight Application in Visual Studio 2008 .................... 52
  The Silverlight Control Project ................................................................. 54
  The Web Project ..................................................................................... 60
  Building a Silverlight 2 Game ................................................................. 62
    Creating the UI in XAML ....................................................................... 62
    Writing the Game Code ....................................................................... 63
  Summary ................................................................................................. 70
Chapter 1
Introducing Silverlight 2

Silverlight represents the next step toward enriching the user’s experience through the technology of the Web. The goal of Silverlight is to bring the same fidelity and quality found in the user interfaces (UIs) associated with desktop applications to Web applications, allowing Web developers and designers to build applications for their clients’ specific needs. It is designed to bridge the technology gap between designers and developers by giving them a common format in which to work. This format will be rendered by the browser and will be based on XML, making it easy to template and to generate automatically. The format is XAML—Extensible Application Markup Language.

Before XAML, a Web experience designer would use one set of tools to express a design using familiar technology. The developer would then take what the designer provided and would interpret it using the technology of his or her choice. The design would not necessarily transfer properly or problem-free into development, and the developer would need to make many alterations that could compromise the design. With Silverlight, the designer can use tools that express a design as XAML, and the developer can pick up this XAML, activate it with code, and deploy it.

Microsoft Silverlight is a cross-browser, cross-platform plug-in that was developed to deliver rich media experience and rich interactive Internet applications via the Web. It offers a full programming model that supports AJAX, .NET, and dynamic languages such as Python and Ruby. Silverlight 1.0 is programmable by using actual Web technologies including AJAX, JavaScript, and DHTML. Silverlight 2 adds dynamic and .NET language support, as well as a host of new features that are only possible when using the .NET Framework, such as Isolated Storage, Networking, a rich control set, and more.

The first part of this book will introduce you to the fundamentals of Silverlight 2 by looking at the design and development tools that are available to you, and the second part will examine the programming model more closely.

Silverlight and User Experience

Silverlight is designed to be part of a much larger ecosystem that is used to deliver the best possible end-user experience. There are a number of typical scenarios for accessing information via the Internet:

- Mobile devices
- Digital home products
Part I Introducing Silverlight 2

- Unenhanced browsers (no plug-ins)
- Enhanced browsers (using plug-ins such as Flash, Java, or Silverlight)
- Desktop applications
- Office productivity software

Over the years, users’ expectations about how these applications should work have evolved. For example, the expectation is that the experience of using an application on a desktop computer should provide more to the user than the same type of application on a mobile device because, as users, we are accustomed to having much more power on the desktop than we do on a mobile device. In addition, many users assume that “because this application is on the Web,” it may not have the same capacity level as a similar desktop application. For example, a user may have lower expectations about a Web-based e-mail application because they don’t believe it can offer the same e-mail capability that office productivity software such as Microsoft Office Outlook provides.

However, as these platforms are converging, the user’s expectations are also increasing—and the term rich is now commonly used to describe an experience above the current baseline level of expectation. For example, the term “rich Internet application” was coined in response to the increased level of sophistication that Web users were seeing in applications powered by AJAX to provide a more dynamic experience in scenarios, such as e-mail and mapping. This evolution in expectations has led to customers who now demand ever richer experiences that not only meet the needs of the application in terms of functionality and effectiveness but also address the perception of satisfaction that the user has with a company’s products and services. This can lead to a lasting relationship between the user and the company.

As a result, Microsoft has committed to the User Experience (UX) and is shipping the tools and technologies that you as a developer can use to implement rich UX applications. Additionally, they are designed to be coherent—that is, skills in developing UX-focused applications will transfer across the domains of desktop and Web application development. So, if you are building a rich desktop application but need a Web version, then you will have a lot of cross-pollination between the two. Similarly, if you are building a mobile application and need an Internet version, you won’t need two sets of skills, two sets of tools, and two sets of developers.

Concentrating on the Web, Figure 1-1 shows the presentation and programming models that are available today. As you can see, the typical browser-based development technologies are CSS/DHTML in the presentation model and JavaScript/AJAX/ASP.NET in the development model. On the desktop, with the .NET Framework 3.x, XAML provides the presentation model, and the framework itself provides the development model. There is an overlap between these, and this is where the Silverlight-enhanced browser provides a “best of both worlds” approach.
The typical rich interactive application is based on technologies that exist in the unenhanced browser category. The typical desktop application is at the other end of the spectrum, using unrelated technologies. The opportunity to bring these together into a rich application that is lightweight and runs in the browser is realized through the Silverlight-enhanced browser that provides the CSS/DHTML and XAML design model and the JavaScript/AJAX/.NET Framework programming model.

Silverlight achieves this by providing a browser plug-in that enhances the functionality of the browser with the typical technologies that provide rich UIs, such as timeline-based animation, vector graphics, and audiovisual media. These are enabled by the Silverlight browser-based XAML rendering engine. The rich UI may be designed as XAML, and because XAML is an XML-based language and because XML is just text, the application is firewall-compatible and (potentially) search-engine friendly. The browser receives the XAML and renders it.

When combined with technology such as AJAX and JavaScript, this can be a dynamic process—you can download snippets of XAML and add them into your UI, or you can edit, rearrange, or remove XAML that is currently in the render tree using simple JavaScript programming.

**Silverlight Architecture**

As I mentioned, the core functionality of Silverlight is provided by a browser plug-in that renders XAML and provides a programming model that can be either JavaScript and browser-based or the .NET Framework and CLR-based. The architecture that supports this is shown in Figure 1-2. When scripting the control in the browser, the main programming interface that is exposed in Silverlight 1.0 is via the JavaScript DOM API. This allows you to catch user events that are raised within the application (such as mouse moves or clicks over a specific element)
and have code to execute in response to them. You can call methods on the JavaScript DOM for XAML elements in order to manipulate them—allowing, for example, control of media playback or animations to be triggered.

For a richer and more powerful experience, you can also program an application that is rendered by the control using the new .NET Framework CLR. In addition to what you can do in JavaScript, this capability offers many of the namespaces and controls that come as part of the .NET Framework, allowing you to do things that are either very difficult—or not possible—in JavaScript, such as accessing data with ADO.NET and LINQ, communicating with Web Services, building and using custom controls, and so on.

Additionally, the presentation runtime ships with the software necessary to allow technologies such as WMV, WMA, and MP3 to be played back in the browser without any external dependencies. So, for example, Macintosh users do not need Windows Media Player to play back WMV content—Silverlight is enough. Underpinning the entire presentation runtime is the presentation code, and this manages the overall rendering process. This is all built into the browser plug-in that is designed to support the major browsers available for both Windows and the Macintosh.

The architecture of a simple application running in the browser using Silverlight is shown in Figure 1-3.
As the application runs within the browser, it is typically made up of HTML. This markup contains the calls to instantiate the Silverlight plug-in. As users interact with the Silverlight application, they raise events that can be captured by either JavaScript or .NET Framework functions. In turn, program code can make method calls against the elements within the Silverlight content to manipulate it, add new content, or remove existing content. Finally, XAML can be read by the plug-in and rendered. The XAML itself can exist inline in the page, externally as a static file, or as dynamic XAML returned from a server.

**Silverlight and XAML**

Now that we’ve taken a high-level look at the architecture of Silverlight and how a typical application will look, let’s examine the base technology that holds the UX together: XAML.

XAML is an XML-based language that is used to define the visual assets of your application. This includes UIs, graphical assets, animations, media, controls, and more. It was introduced by Microsoft for the Windows Presentation Foundation (WPF), formerly Avalon, which is a desktop-oriented technology and part of the .NET Framework 3.0 and beyond. It’s designed, as discussed earlier, to bridge the gap between designers and developers when creating applications.

The XAML used in Silverlight differs from that in the WPF in that it is a subset that is focused on Web-oriented features. So, if you’re familiar with XAML from the WPF, you’ll notice some missing tags and functionality, such as the `<Window>` element.

XAML uses XML to define the UI using XML elements. At the root of every Silverlight XAML document is a container element, such as a `Canvas`, that defines the space on which your UI
will be drawn. When building a Silverlight Web application, you'll have a root Canvas that contains the XML namespace declarations that Silverlight requires.

Here's an example:

```xml
<Canvas
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="640" Height="480"
    Background="White"
>
</Canvas>
```

You will notice that two namespaces are declared. The typical XAML document contains a base set of elements and attributes as well as an extended set, which typically uses the x: prefix. An example of an extended namespace attribute is the commonly used x:Name, which is used to provide a name for a XAML element, allowing you to reference it in your code. The root Canvas element declares the namespace location for each of these.

The Canvas element is a container. This means that it can contain other elements as children. These elements can themselves be containers for other elements, defining a UI as an XML document tree. So, for example, the following is a simple XAML document containing a Canvas that contains a number of children, some of which are Canvas containers themselves:

```xml
<Canvas
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="640" Height="480"
    Background="Black"
>
    <Rectangle Fill="#FFFFFFFF" Stroke="#FF000000"
        Width="136" Height="80"
        Canvas.Left="120" Canvas.Top="240"/>
    <Canvas
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Width="640" Height="480"
        Background="Black"
>
        <Rectangle Fill="#FFFFFFFF" Stroke="#FF000000"
            Width="104" Height="96"
            Canvas.Left="400" Canvas.Top="320"/>
        <Canvas
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            Width="320" Height="104"
            Canvas.Left="96" Canvas.Top="64"
>
            <Rectangle Fill="#FFFFFFFF" Stroke="#FF000000"
                Width="120" Height="96"/>
            <Rectangle Fill="#FFFFFFFF" Stroke="#FF000000"
                Width="168" Height="96"
                Canvas.Left="152" Canvas.Top="8"/>
        </Canvas>
    </Canvas>
</Canvas>
```

Here you can see that the root Canvas has two children, a Rectangle and another Canvas. This second Canvas also contains a Rectangle and a Canvas, and the final Canvas contains two
more Rectangles. This hierarchical structure allows for controls to be grouped together logically and to share common layout and other behaviors.

Silverlight XAML supports a number of shapes that can be combined together to form more complex objects. You’ll find a lot more details about using XAML in Chapter 4, “XAML Basics,” but a few of the basic shapes available include the following:

- **Rectangle** Allows you to define a rectangular shape on the screen
- **Ellipse** Allows you to define an ellipse or circle
- **Line** Draws a line connecting two points
- **Polygon** Draws a many-sided shape
- **Polyline** Draws many line segments
- **Path** Allows you to create a nonlinear path (like a scribble)

In addition, XAML supports brushes, which define how an object is painted on the screen. The inside area of an object is painted using a fill brush, and the outline of an object is drawn using a stroke. Brushes come in many types, including solid color, gradient, image, and video.

Following is an example using a SolidColorBrush to fill an ellipse:

```xml
<Ellipse Canvas.Top="10" Canvas.Left="24"
    Width="200" Height="150">
    <Ellipse.Fill>
        <SolidColorBrush Color="Black" />
    </Ellipse.Fill>
</Ellipse>
```

In this case, the brush uses one of the 141 Silverlight-supported named colors, Black. You also can use standard hexadecimal RGB color notation for custom colors.

Fills and strokes also may have a gradient fill, using a gradient brush. The gradient is defined by using a number of gradient stops across a normalized space. So, for example, if you want a linear gradient to move from left to right—phasing from black to white through shades of gray—you would define stops according to a normalized line. In this case, consider the beginning of the normalized line as the 0 point and the end as the 1 point. So, a gradient from left to right in a one-dimensional space has a stop at 0 and another at 1. Should you want a gradient that transitions through more than two colors—from black to red to white, for example—you would define a third stop somewhere between 0 and 1. Keep in mind that when you create a fill, however, you are working in a two-dimensional space, so (0,0) represents the upper-left corner, and (1,1) represents the lower-right corner. Thus, to fill a rectangle with a gradient brush, you would use a LinearGradientBrush like this:

```xml
<Rectangle Width="200" Height="150">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="Black" Offset="0" />
            <GradientStop Color="Red" Offset="0.5" />
            <GradientStop Color="White" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```
XAML also supports text through the TextBlock element. Control over typical text properties such as content, font type, font size, wrapping, and more are available through attributes. Following is a simple example:

```xml
<TextBlock TextWrapping="Wrap" Width="100">
  Hello there, how are you?
</TextBlock>
```

Objects can be transformed in XAML using a number of transformations. Some of these include the following:

- **RotationTransform**  Rotates the element through a defined number of degrees
- **ScaleTransform**  Used to stretch or shrink an object
- **SkewTransform**  Skews the object in a defined direction by a defined amount
- **TranslateTransform**  Moves the object in a direction according to a defined vector
- **MatrixTransform**  Used to create a mathematical transform that can combine all of the above

Transformations may be grouped so that you can provide a complex transformation by grouping existing ones. That is, you could move an object by translating it, change its size by scaling it, and rotate it simultaneously by grouping the individual transformations together. Here’s a transformation example that rotates and scales the canvas:

```xml
<Canvas.RenderTransform>
  <TransformGroup>
    <RotateTransform Angle="-45" CenterX="50" CenterY="50"/>
    <ScaleTransform ScaleX="1.5" ScaleY="2"/>
  </TransformGroup>
</Canvas.RenderTransform>
```

XAML supports animations through defining how their properties are changed over time using a timeline. These timelines are contained within a storyboard. Different types of animation include:

- **DoubleAnimation**  Allows numeric properties, such as those used to determine location, to be animated
- **ColorAnimation**  Allows colored properties, such as fills, to be transformed
PointAnimation  Allows points that define a two-dimensional space to be animated

As you change properties, you can do it in a linear manner, so that the property is phased between values over a timeline, or in a "key frame" manner, in which you would define a number of milestones along which the animation occurs. We'll examine all of this in a lot more detail in Chapter 5, "XAML: Transformation and Animation.”

Beyond this basic XAML, you will define your full UIs using controls and layout using XAML, too. These will be explored in more detail in Chapter 7, “Silverlight Controls: Presentation and Layout,” and in the rest of the chapters in Part 2 “Programming Silverlight 2.”

Silverlight and the Expression Suite

Microsoft has introduced the Expression Suite of tools to provide a robust, modern set of tools for designers to express their work using artifacts that developers can include while developing using the Microsoft Visual Studio tool suite.

There are several tools in the Expression Suite:

• **Expression Web**  This is a Web design tool that allows you to use HTML, DHTML, CSS, and other Web standard technologies to design, build, and manage Web applications.

• **Expression Media**  This is a media asset management tool that permits you to catalog and organize these assets, including the facility to encode and change encoding between different formats.

• **Expression Encoder**  This application is designed to allow you to manage encoding of media assets. It can also be used to bundle media with the relevant code to have a Silverlight media player for it.

• **Expression Design**  This is an illustration and graphic design tool that you can use to build graphical elements and assets for Web and desktop application UIs.

• **Expression Blend**  This tool is designed to let you build XAML-based UIs and applications for the desktop with WPF or for the Web with Silverlight.

When using Silverlight, you’ll use some or all of these applications. In the rest of this chapter, we’ll take a look at how Design, Blend, and Encoder enhance your toolkit in designing and building Silverlight applications.

Silverlight and Expression Design

Expression Design is a graphical design tool that allows you to build graphical assets for use in your applications. It’s a huge and sophisticated tool, so we will just provide an overview of how it can be used for Silverlight XAML here. Expression Design allows you to blend vector-based and raster-based (bitmap) images for complete flexibility.
It supports many graphical file formats for import, such as:

- Adobe Illustrator—PDF Compatible (*.ai)
- Adobe Photoshop (*.psd)
- Graphical Interchange Format (.gif)
- Portable Network Graphics format (.png)
- Bitmaps (.bmp, .dib, .rle)
- JPEG formats (.jpeg, .jpg, .jpe, .jfif, .exif)
- Windows Media Photos (.wdp, .hdp)
- Tagged Image File Format (.tiff, .tif)
- Icons (.ico)

It supports export of the following image types:

- XAML Silverlight Canvas
- XAML WPF Resource Dictionary
- XAML WPF Canvas
- Adobe Illustrator (.ai)
- Portable Document Format (.pdf)
- Adobe Photoshop (.psd)
- Tagged Image File Format (.tiff, .tif)
- JPEG formats (.jpeg, .jpg)
- Windows Bitmap (.bmp)
- Portable Network Graphics format (.png)
- Graphical Interchange Format (.gif)
- Windows Media Photos (.wdp)

As you can see, Expression Design supports export of graphical assets as XAML files. Later in this chapter, you’ll see how to use Expression Design to design the graphical elements of a simple application, and you’ll export these as XAML, which you can use in Expression Blend and Visual Studio to create an application.

Figure 1-4 shows the Export XAML dialog box in Expression Design. There are several format options, one of which is XAML Silverlight Canvas (shown selected). This option will format your drawing using the subset of XAML elements that are usable by Silverlight, allowing you to import the resulting XAML into Visual Studio or Expression Blend to build your Silverlight application.
FIGURE 1-4 Exporting XAML from Expression Design.

This will export the content as an XML document containing a Canvas element that contains the elements of your design. Here's a (truncated) example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Canvas xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" x:Name="Document">
    <Canvas x:Name="Layer_1" Width="640.219" Height="480.202" Canvas.Left="0" Canvas.Top="0">
        <Ellipse x:Name="Ellipse" Width="135" Height="161" Canvas.Left="0.546544"
            Canvas.Top="20.3998" Stretch="Fill" StrokeLineJoin="Round" Stroke="#FF000000"
            Fill="#FFFFC800"/>
        <Path x:Name="Path" Width="135.103" Height="66.444" Canvas.Left="-0.555986"
            Canvas.Top="-0.389065" Stretch="Fill" StrokeLineJoin="Round" Stroke="#FF000000"
            Fill="#FF000000" Data="..."/>
        <Path x:Name="Path_0" Width="19.4583" Height="23.9019" Canvas.Left="75.8927"
            Canvas.Top="76.1198" Stretch="Fill" StrokeLineJoin="Round" Stroke="#FF000000"
            Fill="#FF000000" Data="..."/>
        <Path x:Name="Path_1" Width="11.0735" Height="24.0564" Canvas.Left="60.473"
            Canvas.Top="106.4" Stretch="Fill" StrokeLineJoin="Round" Stroke="#FF000000"
            Fill="#FF000000" Data="..."/>
        <Path x:Name="Path_2" Width="76" Height="29.8274" Canvas.Left="31.5465"
            Canvas.Top="127.4" Stretch="Fill" StrokeThickness="7" StrokeLineJoin="Round"
            Stroke="#FF000000" Data="..."/>
        <Path x:Name="Path_3" Width="20.3803" Height="27.1204" Canvas.Left="31.2028"
            Canvas.Top="75.306" Stretch="Fill" StrokeLineJoin="Round" Stroke="#FF000000"
            Fill="#FF000000" Data="..."/>
    </Canvas>
</Canvas>
```
You can then cut and paste this XAML into Expression Blend or Visual Studio, and you will be able to use the graphical element in your application.

**Silverlight and Expression Blend**

Expression Blend has native support for the creation of Silverlight applications. When you launch Expression Blend and create a new project, you have two options for creating Silverlight projects, as you can see from Figure 1-5.

![FIGURE 1-5 Silverlight support in Expression Blend.](image)

The two options for Silverlight projects are:

- **Silverlight 1 Site** This creates a Silverlight JavaScript project, giving you a folder that contains a simple Web application containing an HTML page that has the requisite scripts to embed a Silverlight object as well as a default XAML document containing a single canvas. It does not contain any of the implementation details for .NET programming, so the descriptive term 1 Site is used, even though the Silverlight control is still version 2. This will likely change in future versions of Expression Blend to Silverlight JavaScript Site. Chapter 6, “The Silverlight Browser Control,” will look at programming JavaScript applications in a little more detail.

- **Silverlight 2 Application** This creates a Silverlight project with everything necessary to program against it using the .NET Framework. There will be more on Silverlight 2 in Chapter 3, “Using Visual Studio with Silverlight 2,” and then Part 2 of this book (Chapters 7–14) will cover it in much more detail.

**Exploring the Silverlight 1 Site Project**

When you create a new Silverlight Script application using Blend, your project will contain a default HTML file that contains all the requisite JavaScript to instantiate the Silverlight control.
In addition, Blend also creates a basic XAML page called Page.xaml and an associated JavaScript file called Page.xaml.js. Expression Blend treats this as a “code-behind” JavaScript file in a manner that is similar to how Visual Studio treats the C# code-behind file associated with an ASPX page. Finally, Blend gives you a copy of the Silverlight.js file that is part of the Silverlight software development kit (SDK). This file manages the instantiation and downloading of the Silverlight plug-in for your users. You can see the project structure in Figure 1-6.

![Project structure for a Silverlight Script application.](image)

**FIGURE 1-6** Project structure for a Silverlight Script application.

### The Default Web Page

Listing 1-1 shows the code for the basic Web page that is created for you by Blend for Silverlight projects.

**LISTING 1-1** Default.html from Silverlight Template

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!-- saved from url=(0014)about:internet -->
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>SilverlightSite1</title>
<script type="text/javascript" src="Silverlight.js"></script>
<script type="text/javascript" src="Page.xaml.js"></script>
<style type="text/css">
#silverlightControlHost {
  height: 480px;
  width: 640px;
}
#errorLocation {
  font-size: small;
  color: Gray;
}</style>
<script type="text/javascript">
function createSilverlight()
{
  var scene = new SilverlightSite1.Page();
  Silverlight.createObjectEx({
```
```
As you can see, it imports two JavaScript files: Silverlight.js and Page.xaml.js. You’ll be looking at each of these files shortly.
Chapter 1  Introducing Silverlight 2

The Silverlight control instantiation takes place in the `<div>` at the bottom of the page. This contains a call to the `createSilverlight` function, which is implemented at the top of the page. This creates a new Silverlight object using either the `createObjectEx` function (which, in turn, resides in Silverlight.js) or the `createObject` function. When using the `createObjectEx` function, the syntax for specifying the parameters uses the JavaScript Object Notation (JSON) syntax, as shown in this example. You can alternatively use the `createObject` function, which uses standard parameters.

The first parameter is the source XAML. This can be a reference to a static external file (which is used in this case as `Page.xaml`), a reference to the URL of a service that can generate XAML, or a reference to a named script block on the page that contains XAML.

The second parameter is the parent element. This is the name of the `<div>` that contains the Silverlight control. As you can see in Listing 1-1, this is called `SilverlightControlHost`.

The third parameter is the ID that you want to use for this control. If you have multiple Silverlight controls on a page, you need to have a different ID for each.

The fourth parameter is the property settings for the control properties. These can include simple properties such as width, height, and background color, as well as complex ones. More complex property settings include:

- `inplaceInstallPrompt`  Determines the install type for Silverlight. If this is set to `true`, the user implicitly accepts the license and directly downloads and installs the plug-in. If it is set to `false`, the user is directed to `http://www.silverlight.net` and, from that site, can accept the license and download the plug-in.
- `isWindowless`  If set to `true`, the control is considered `windowless`, meaning that you can overlay non-Silverlight content on top of it.
- `framerate`  Determines the maximum frame rate for animations.
- `version`  Determines the minimum Silverlight version your application will accept. As you can see in Listing 1-1, the version is listed as 1.0—this isn’t a bug, but simply an instruction that this application is backward compatible and should work on 1.0. If this was instead 2.0, and Silverlight 2 was not installed, then the user would be taken to the install experience for Silverlight 2.

The fifth parameter is used to map events to event handlers. The events are implemented in a JavaScript class called `scene`, which was declared at the top of the function:

```javascript
var scene = new SilverlightSite1.Page();
```
Part I Introducing Silverlight 2

The `createSilverlight` function declares that the `onLoad` event should be handled by a member function of the `scene` class called `scene.handleLoad`. It does this by creating a delegate using this syntax:

```
onload: Silverlight.createDelegate(scene, scene.handleLoad)
```

This class is implemented in the JavaScript code-behind for Page.xaml called Page.xaml.js. You can see this in Listing 1-2.

**Listing 1-2** JavaScript Code-Behind Page.xaml

```javascript
if (!window.SilverlightSite1)
    SilverlightSite1 = {};

SilverlightSite1.Page = function()
{
}

SilverlightSite1.Page.prototype =
{
    handleLoad: function(control, userContext, rootElement)
    {
        this.control = control;

        // Sample event hookup:
        rootElement.addEventListener("MouseLeftButtonDown",
            Silverlight.createDelegate(this, this.handleMouseDown));
    },

    // Sample event handler
    handleMouseDown: function(sender, eventArgs)
    {
        // The following line of code shows how to find an element by name
        this.control.content.findName("Storyboard1").Begin();
    }
}
```

Here you can see JavaScript code to create a class called `SilverlightSite1.Page`. It contains two member functions, `handleLoad` and `handleMouseDown`.

The function `handleLoad` adds another event listener for the `MouseLeftButtonDown` event by creating a delegate associating this event and the `handleMouseDown` function, which is also defined within this JavaScript.

Thus, the template application creates a default HTML file that contains an instance of Silverlight with a single canvas that fires an event when it loads. The load event wires up the mouse down event, demonstrating that event declaration, delegation, and handling are available at both design time and run time.
Chapter 1  Introducing Silverlight 2

Silverlight and Expression Encoder

Expression Encoder is an application that allows you to encode, enhance, and publish your video content using Silverlight. It comes with a UI that is consistent with the rest of the Expression suite or with a command-line interface that can be used for batch work. You can see Expression Encoder in Figure 1-7.

![Expression Encoder](image)

**FIGURE 1-7** Expression Encoder.

Expression Encoder allows you to import video from any format for which a DirectShow filter is available and installed in your system. It will then re-encode the video into a VC-1–capable WMV using one of a number of preset profiles optimized for the delivery client. These include settings for devices, as well as for streaming or on-demand content delivered over the Internet.

You aren’t limited to what the preset profiles give you—you can override any of the video and audio encoding settings. Figure 1-8 (on page 20) shows an example of how a video encoding may be tweaked.

Media Encoder includes a number of preset media player applications for Silverlight. These will "wrap up" your video with a Silverlight JavaScript-based application that can be used on any Web server to provide a complete Silverlight-based viewing experience.

In addition to encoding, metadata can be added to your video. A classic metadata experience is when tags are encoded into the video and the application then reacts to these tags. Inserting tags with Expression Encoder is very simple. Simply drag the playhead to the desired point, select Add Marker, and enter the appropriate information for the marker.
You can see this in Figure 1-9 on the right side of the screen, where the marker time and type of ball that is shown on the screen at that time has been configured.
The Output tab allows you to select the template player that you want to use.

Figure 1-10 shows where the template that matches the Expression product line has been selected. To create a video player with this template, simply import a video, and press the Encode Button with this template selected.

![Figure 1-10 Using Encoder to build a Silverlight media player.](image)

After you've done this, you'll get a full-featured media player in Silverlight for your video content. You can see an example of a Silverlight media player in Figure 1-11.

This section just scratches the surface of what is possible with Expression Encoder and how it can be used with Silverlight. For more details, please refer to [http://www.microsoft.com/expression](http://www.microsoft.com/expression).
Part I Introducing Silverlight 2

Summary

In this chapter, you were introduced to Silverlight 2 and learned how it fits into the overall Web and UX landscape. You discovered how technology from Microsoft is applied to current UX scenarios, and you were introduced to an overview of the Silverlight architecture, including XAML and how it is used to implement rich UIs.

Additionally, you saw how the Microsoft Expression Suite is designed to complement traditional development tools such as Visual Studio for creating Silverlight applications. You specifically learned how Expression Design is used to build graphical assets and how Expression Blend is used to link these together into an interactive application as well as using Expression Encoder to manage your video assets.

Now it’s time to go deeper. In the next few chapters, you’ll learn more about the Silverlight API, starting with a more detailed examination of Expression Blend and how it is used by Silverlight in the next chapter.
Chapter 2
Using Expression Blend with Silverlight 2

Expression Blend is a professional design tool intended to create engaging experiences for Windows and the Web. It allows you to blend all the necessary design elements for your Web experiences, including video, vector art, text, animation, images, and other content such as controls, with one set of tools. Expression Blend is designed to aid you in the building of Windows-based as well as Web-based applications. This chapter will introduce you to this tool, giving you a tour of what is possible with it. Expression Blend has far too many aspects to cover in one chapter, but by the end of this chapter, you’ll have a good grasp of the basics and will be ready to delve further into the features of this wonderful tool on your own!

Getting Started with Expression Blend

Expression Blend is available as part of the Microsoft Expression suite. Details are available at http://www.microsoft.com/expression.

After you’ve downloaded and installed Expression Blend, launch it from the Start menu. You’ll see the Blend integrated development environment (IDE), as shown in Figure 2-1.

To create a new application, select New Project from the File menu to open the New Project dialog box, as shown in Figure 2-2.
The options that you are given are:

- **WPF Application (.exe)**  This option creates a client-executable application built on the Windows Presentation Foundation (WPF); this type of project is a *Windows Only* application.

- **WPF Control Library**  This option creates a DLL file that may be used for shared controls across WPF applications; this type of project is a *Windows Only* application.

- **Silverlight 1 Site**  This option creates a Web site that uses the Silverlight control. It contains the basic JavaScript components to instantiate a Silverlight control as well as a sample XAML document with JavaScript-based event handlers, and it was covered in Chapter 1, “Introducing Silverlight 2.” This option creates a *Web*-based and therefore *multiplatform* application.

- **Silverlight 2 Application**  This option creates a Silverlight application based on the Silverlight 2 runtime. This application includes the .NET Framework runtime that supports your .NET-based applications, allowing them to run in a browser. When you select this type of project, you’ll be able to pick your preferred programming language (either Microsoft Visual Basic or Microsoft Visual C#). This option also creates a *Web*-based and therefore *multiplatform* application.

Chapter 1 gave you some details about the Silverlight 1 Site template, and you can refer back to that chapter for more information about this option. For the rest of this chapter, we’ll be focusing on the Silverlight 2 Application template.
Creating a Silverlight 2 Application

Open the New Project dialog box to create a Silverlight 2 application, and name your new project TestApp. Expression Blend will create a new project for you that contains everything you need for a Silverlight 2 .NET application.

You can see the project structure that it creates in Figure 2-3. This is identical to the product structure that is built by Visual Studio, which will be discussed in much more detail in Chapter 3, “Using Visual Studio with Silverlight 2.”

What’s important to note about the structure is that there are two XAML files in this application, and neither of them are a Silverlight XAML page, as in Silverlight 1. This is one way in which a Silverlight 2 Application project differs fundamentally from a Silverlight 1 Site project.

The Default Page

Silverlight 2 deals with your XAML as a Page. Thus, the template creates your default application XAML content as a file named Page.xaml. You’ll see that the root of this, not surprisingly, is a UserControl and not a Canvas, as you may have been familiar with from Silverlight 1.
Following is the XAML for the default Page.xaml:

```xml
<UserControl
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    x:Class="TestApp.UserControl1"
    d:DesignWidth="640" d:DesignHeight="480">
    <Grid x:Name="LayoutRoot" Background="White" />
</UserControl>
```

Note the use of `<UserControl>` to host the content.

You'll see a small difference between this XAML and that created by Microsoft Visual Studio (in Chapter 3). For example, this XAML has a few extra namespace declarations. Blend uses these for parsing the XAML to render in the designer. They don't affect your design beyond that, and you can safely ignore these.

You'll see that UserControl1 has a code-behind file that is generated for you. This will be named UserControl1.xaml.cs or UserControl1.xaml.vb, depending on which language you selected when creating the project. The file contains the basic code required to construct the `UserControl`. You can see it here:

```csharp
namespace TestApp
{
    public partial class UserControl1 : UserControl
    {
        public UserControl1()
        {
            // Required to initialize variables
            InitializeComponent();
        }
    }
}
```

You aren't restricted to using this file for your application logic. You can, of course, create other .cs (or .vb) files that can contain shared logic, but this one will be launched whenever the control is instantiated by the Silverlight runtime.
The Default App.xaml and Code-Behind Files

App.xaml and App.xaml.cs define the startup conditions for your application. These will be the first things loaded and executed by Silverlight on startup and the last things closed when the application is shut down.

This is accomplished using the OnStartup and OnExit events. These are set up for you by the project template. Note that UserControl1 does not render by default—it has to be instructed to render as part of the applications startup. This is accomplished in the OnStartup event handler, where the RootVisual for the application is set to an instance of UserControl1:

```csharp
public App()
{
    this.Startup += this.OnStartup;
    this.Exit += this.OnExit;
    InitializeComponent();
}

private void OnStartup(object sender, StartupEventArgs e)
{
    // Load the main control here
    this.RootVisual = new Page();
}

private void OnExit(object sender, EventArgs e)
{
}
```

App.xaml does not support visual elements directly, so you cannot add controls or other visual elements directly. Just because it is XAML, don’t think of it as a design surface. In this case, XAML is used for definition purposes only. For example, you can define application-specific resources for your application using it.

App.xaml.cs is useful for initialization of data that you want to use across several user controls. Keep this in mind as you design your application. For example, you could store some text that could be used across your application by declaring it as a resource in your App.xaml:

```xml
<Application
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="TestApp.App">
    <Application.Resources>
        <TextBlock x:Key="txtResource" Text="Hello"/>
    </Application.Resources>
</Application>
```

You can now easily access this content from any control in your application as follows:

```csharp
TextBlock t = (TextBlock)Application.Current.Resources["txtResource"]; string strTest = t.Text;
```
Introducing Silverlight 2

Executing the Application

One thing that you may see is missing if you are sharp-eyed is a page to host the Silverlight control. Don't worry! Blend will automatically generate one for you. You'll see this when you launch the application.

Before going any further, add a simple TextBlock to your UserControl to render some text. Here’s an example:

```xml
<UserControl
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    x:Class="TestApp.UserControl1"
    d:DesignWidth="640" d:DesignHeight="480">
    <Grid x:Name="LayoutRoot" Background="White">
        <TextBlock Text="Hello"/>
    </Grid>
</UserControl>
```

Now if you execute the application, you’ll see something like the output shown in Figure 2-4.

![Figure 2-4 Running the Silverlight application from Blend.](image)

This simple application runs using the Cassini Web server (hence the random port number, 55924, that you can see in the address box in Figure 2-4) by generating an HTML page to host the Silverlight content.

Let's take a look at the source code for this page by using the browser command View Source. You can see the source code here:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

Do take note of the `<object>` tag. This attempts to instantiate Silverlight, and should it fail, it renders an image with a hypertext reference (HREF) to the Silverlight download in its place. You’ll see more about this and other ways of instantiating the Silverlight object in Chapter 6, “The Silverlight Browser Control.”
The Expression Blend IDE

Expression Blend offers a flexible IDE that is designed to maximize the amount of information on the screen while keeping it easy for the user to understand what is going on and not be overwhelmed.

The IDE has two main application workspace layouts: the Design workspace, which is used primarily for constructing and customizing your user interface (UI), and the Application workspace, which is used primarily for designing your timeline-based animations. You can switch between the workspaces using the F6 key or by selecting the workspace you want from the Active Workspace options on the Window menu.

The screen is divided into *panes* in the Expression Blend IDE, and each of the panes has a fixed purpose, as you'll discover when we tour them now.

**The Tools Pane**

The tools pane is on the far left side of the screen. It contains *tools*, such as Paint or Clip, that can be used to manipulate any object; *visual elements*, such as a *Rectangle* or *Ellipse*; *layout elements*, such as the *StackPanel* or *Canvas*; and *controls*, such as *Button* or *TextBox*). You can see the tools pane in Figure 2-5.

![Figure 2-5 Expression Blend tools pane.](image)

In Blend, similar tools can be collected together into a single icon on this tool pane. If you look at Figure 2-5, you can see how to view a set of similar tools by finding the white triangle
in the lower-right corner of the tool. When this triangle is present, you can hold down the mouse button on that tool to find more members in the same “family” as the selected object. So, for example, if you hold down the mouse on the Rectangle tool, you’ll see a pop-up box that shows you the other available shapes, as you can see in Figure 2-6.

One nice shortcut that Blend provides is the way it creates a default tool on the toolbar when you have used a tool from the family of tools. That is, the tool that you just used will be displayed on the toolbar, so you don’t need to hold down the mouse, wait for the menu, and then select the tool again to use it the next time.

So, for example, in Figure 2-6, the Rectangle is displayed on the toolbar, and when you hold down the mouse, you will see a box displaying the other visual element tools of this type that are available. If you then select the Ellipse and draw with it on the design surface, the toolbar will change to display the Ellipse instead of the Rectangle.

The Interaction Pane

The interaction pane, shown in Figure 2-7 and usually located just to the right of the tools pane, is designed to help you with the following tasks:

- View all of the objects on your design surface, including their hierarchy when you are using container objects.
- Select objects so you can modify them. This isn’t always possible on the design surface because objects can be placed off screen or behind other objects.
- Create and modify animation timelines. You’ll learn more about how to do this in the section titled “Using Blend to Design Animations” later in this chapter.

The interaction pane is designed to have two separate highlights. The currently selected object is highlighted in grey—in Figure 2-7, you can see that the Border control is highlighted in grey. This is the object that you can currently amend with the properties window or by dragging it around the design surface.

Although it appears grey in the figure, you can see on your screen that the LayoutRoot control has a yellow border around it. On the design surface, you’ll also see this yellow border. This indicates that this is the currently selected container.

In addition to manipulating objects, you also use the interaction pane to create animations and storyboards. You do this by clicking on the plus sign (+) button at the top of the interac-
introduction pane. You’ll explore the ways you can use this to create animations in the section titled “Using Blend to Design Animations” later in this chapter.

FIGURE 2-7 Interaction pane.

The **Design Surface**

The design surface is the main pane in the Expression Blend IDE screen, and this is where you can manipulate all the objects visually or by amending their underlying XAML code directly.

On the right side of the design pane, you will see three tabs:

- The Design tab gives you the pure design surface.
- The XAML tab gives you the XAML editing Window.
- The Split tab provides you with a split window—one half in design view and the other half in XAML view.

You can see the design pane in split view in Figure 2-8.
Note that you can use the Zoom feature in design view, so when you are working on sophisticated interfaces, you can zoom in for a detailed view and zoom out for an overview. You do this using the Zoom tool at the lower-left corner of the design pane. You can drop it down to select preset zoom settings, type the specific value you want in the box provided, or drag the mouse within it to set the desired zoom level.

The Project Pane

The project pane (shown in Figure 2-9) is used to manage the files in your project. The important thing to note in this pane is the use of context menus. Depending on where you right-click in this pane, you’ll get a different (and appropriate) context menu. You might be familiar with context menus that provide commands for a specific pane, but in this case, you’ll get different menus when you right-click the solution, the project, the References folder, and so on in the project pane.
A solution is a collection of one or more projects. When you edit a solution, you can manage everything to do with the solution itself, including building, debugging, cleaning, and managing individual projects. In Figure 2-9, you can see the solution TestApp listed at the top of the project pane, and the pane indicates that there is one project within the solution.

A project is a collection of items that, when combined, make up an application that contains one or more Silverlight Pages. The project definition contains all the references to external components that this application needs within the References folder. When you right-click the project, the context menu that displays for the project allows you to manipulate the contents of the project, with options such as adding new items based on a template, adding existing items from other projects, or deleting items from the project.

The References folder within the project is used to manage references to precompiled assemblies that contain information that you want to use in your project. For example, if you want to use a custom control, it will be compiled into an assembly, so if you reference that assembly in your references, you can then use it within your application.

The Properties folder contains the application manifest file that describes all the properties of the project, including the list of references, so that the application can understand from where they are loaded at run time. The Properties folder should not be confused with the properties pane, indicated by the Properties tab at the top of the window shown in Figure 2-9 and explained in more detail in the following section.
The Properties Pane

The properties pane is used to manage all the visual aspects of a particular element. Since XAML elements have many configurable properties, this pane gives you two very useful shortcuts.

The first shortcut is provided by the division of the properties pane into several classifications, typically providing access to the following visual aspects of elements:

- **Brushes**  Allow you to set fill and stroke options as well as use an opacity mask on your element. You’ll see a lot more detail about how brushes are used in Chapter 4, “XAML Basics.”

- **Appearance**  Allows you to set extended appearance properties for your object. Note that the available appearance properties will change drastically based on the object that you are currently editing. So, for example, if you are editing a *Rectangle* element, the Appearance section of the properties pane will allow you to set things like the corner radii, but if you are editing a *Button* element that doesn’t have corner radii, you will not have this option available.

- **Layout**  Allows you to edit the various layout options for your object, such as Width, Height, and Alignment options. You can also use layout options to change the position of an object within a grid—if the layout is on a grid.

- **Common Properties**  Effectively the properties that are common across a *type* of object. So, for example, the common properties for controls that are distinct from shapes are typically edited here. These options can be very difficult to use, depending on the object that you are editing. For example, if you are editing a control, a common property will be its tab index, but if you are editing a shape, the tab index will not be available.

- **Transform**  Provides you with the ability to edit the *RenderTransform* of your object. This defines how the object can be manipulated by the rendering system. Transformations are covered in detail in Chapter 5, “XAML: Transformation and Animation.”

- **Miscellaneous**  The catch-all location for properties that aren’t available on any of the other classifications.

Do take note that these classification panes are further subdivided. You’ll notice that many of them have an arrow at the bottom of the pane that can be used to expand and contract the properties view. This allows you to hide lesser-used properties until you need them.

The second shortcut in the properties pane is its Search feature, which allows you to search for a particular property. For example, if you know you want to edit some features of a font but don’t know the name of the property itself, you can type *font* into the search engine, and the classifications and available properties will be filtered so that only those that have to do with fonts are displayed. This is done immediately upon a keystroke, so if you are searching for a font property—in our example, as soon as you type *fo*—you will see available properties dis-...
played such as `foreground` and `rendertransform` as well as the font properties, as shown in the list of properties displayed at the bottom of Figure 2-10.

![Properties Pane](image)

**FIGURE 2-10** Using the properties pane.

Now let’s take a look at how you can use all these tools we’ve introduced to build Silverlight applications.

**Using Blend to Build Silverlight Applications**

The main design-oriented functions that you can use Blend to accomplish as you put together your application include the following:

- Organizing the layout
- Placing and customizing visual elements
- Placing and customizing controls
- Designing animations

You’ll explore each of these functions of Blend in the rest of this chapter.

**Layout**

In Silverlight, you use special tools to create and organize the layout of your application. There are several options available to you, and we will look at each of them in turn.
Using a Grid

The Grid layout element allows you to lay elements out in a structure that looks like a table. (Do not confuse the Grid layout element with a Grid control that gives you functionality similar to a spreadsheet.) When using a Grid layout tool, you can specify how your elements are placed by indicating their coordinates with virtual row and column designations within the Grid layout. For example, consider the following XAML:

```xml
<Grid x:Name="LayoutRoot" Background="White">
    <Button Height="38" Margin="104,72,0,0" Width="58" Content="Button"/>
    <Button Height="24" Margin="210,72,0,0" Width="54" Content="Button"/>
    <Button Height="49" Margin="0,96,158,0" Width="80" Content="Button"/>
    <Button Height="54" Margin="297,185,270,0" Width="67" Content="Button"/>
    <Button Height="33" Margin="104,217,0,213" Width="87" Content="Button"/>
</Grid>
```

When rendered, this will appear as shown in Figure 2-11.

![Random buttons.](image)

**FIGURE 2-11** Random buttons.

Now, if you wanted to organize these buttons, you could carefully set their positions by dragging them around the design surface to place them at roughly the positions where you want them, but if you position them this way, you will need to zoom in to make sure pixels are aligned.

Alternatively, you could use the Grid layout, where you can use the layout properties of the button to determine its location in the grid. If you start with a new Silverlight project, you'll see that it has a Grid layout element on it called LayoutRoot. Select this element in your project, and look at the Layout properties associated with it. Expand the properties viewer until you see the settings for the ColumnDefinitions and RowDefinitions, as shown in Figure 2-12.
Because ColumnDefinitions and RowDefinitions are collections, each one has an ellipsis (…) button to the right of the setting name. This indicates that another dialog box will open when you click it. Select the button next to the ColumnDefinitions property setting, and the ColumnDefinition collection editor will display, as shown in Figure 2-13.
Use this dialog box to add, remove, and manage columns. Click the Add Another Item button three times to add three columns. Repeat this for the RowDefinitions property setting so that you have a grid that is comprised of three rows and three columns. After you have made these changes to ColumnDefinitions and RowDefinitions, you will see that the designer pane displays a $3 \times 3$ layout grid, as you can see in Figure 2-14.

![The 3 × 3 Layout grid.](image)

FIGURE 2-14 The $3 \times 3$ Layout grid.

Now, whenever you are placing an element on the screen, you’ll see pink guidelines that show you how you can snap to a particular grid element, as shown in Figure 2-15. (They appear as wider grey lines in the figure.) Snapping the button to the grid and column layout like this will ensure that the button is always at that relative position and size in the grid.

Place another button in the central square on the grid, as shown in Figure 2-15. This time, do not snap it to the grid. Then run the application and experiment with resizing the window. You’ll see that the first button will always remain at the same relative position and the same size, but the second button will change its width and/or height to stay relative to the size of the screen.
Using Canvas

The Canvas layout is a completely free-format drawing surface. You can specify the desired location for a control by setting its Canvas.Top and Canvas.Left properties or by using its Margin property.

So, for example, consider the following XAML:

```xml
<Canvas Height="261" Width="439">
  <Button Height="101" Width="110" Canvas.Left="101" Canvas.Top="82.5" Content="Button"/>
</Canvas>
```

You will see that the Canvas.Top and Canvas.Left properties for the button have been set. These indicate that the button will always be at those values relative to the parent Canvas, so as the Canvas moves, the button will move also. The Canvas layout is covered in more detail in Chapter 4.

Using StackPanel

The StackPanel layout will always orient its child controls either horizontally or vertically, stacking them (hence the name) based on the Orientation property. Note that the panel will override the positioning of the controls. For example, look at the following XAML:

```xml
<StackPanel Height="337" Width="224">
  <Button Canvas.Top="100" Height="64" Width="98" Orientation="Vertical" Content="Button"/>
  <Button Height="85" Width="92" Content="Button"/>
</StackPanel>
```
You can see that the first button has its `Canvas.Top` property set to 100. You would expect that this would mean that the control would then be drawn at that position, but as Figure 2-16 shows, this is not the case, and it is stacked by the `StackPanel` layout at the top of the `StackPanel` (because the `StackPanel` has its `Orientation` property set to `Vertical`).

![Figure 2-16 Buttons in a StackPanel.](image)

When you have many controls in a `StackPanel`, you may go beyond the bounds of the Panel control itself, in which case the controls will be clipped to the bounds of the `StackPanel`. To get around this problem, you can use a `ScrollViewer`, which is explained in the next section.

**Using the ScrollViewer**

The `ScrollViewer` provides scroll bars that allow the user to pan around the contents of a layout if the contents exceed the bounds of the `ScrollViewer`. It can only contain one child control, so unless you are using a control that needs a large view area (such as an `Image`), it is typically only used to contain other containers.

For example, following is a `StackPanel` in which the contents exceed the vertical space available to it:

```xml
<StackPanel Height="300" Width="199">
    <Button Height="44" Width="86" Content="Button"/>
    <Button Height="57" Width="75" Content="Button"/>
    <Button Height="70" Width="59" Content="Button"/>
    <Button Height="109" Width="95" Content="Button"/>
    <Button Height="104" Width="88" Content="Button"/>
</StackPanel>
```

The `StackPanel` in this example is 300 pixels high, but the total height of all the buttons is 384 pixels, and so the bottom button will be cropped, as you can see in Figure 2-17.
FIGURE 2-17 Cropped elements in a StackPanel.

Now, if you contain this within a ScrollViewer, you'll get better results. Note that the StackPanel will still crop the buttons if you do not change its height, so if you need to have an area of height 300, you can set the ScrollViewer to have this height and then set the StackPanel to have a different height. Here's the XAML to do this:

```xml
<ScrollViewer Height="300" Width="300">
  <StackPanel Height="400" Width="199">
    <Button Height="44" Width="86" Content="Button"/>
    <Button Height="57" Width="75" Content="Button"/>
    <Button Height="70" Width="59" Content="Button"/>
    <Button Height="109" Width="95" Content="Button"/>
    <Button Height="104" Width="88" Content="Button"/>
  </StackPanel>
</ScrollViewer>
```

You can see how the ScrollViewer created here appears in Figure 2-18.

FIGURE 2-18 Using the ScrollViewer.
Now you can scroll up and down the button list, and the buttons will all be available; none are unavailable because of cropping if you use the ScrollViewer. Note that the button at the bottom in Figure 2-18 can be revealed by dragging the scroll bar down.

**The Border Control**

Not to be confused with the Border Patrol—part of the Department of Homeland Security—the Border Control is simply used to draw a border, background, or both around another element. For example, consider the following XAML:

```xml
<BORDER Height="318" Width="405" Background="#FFFF0000">
  <BUTTON Height="234" HorizontalAlignment="Center" VerticalAlignment="Center" Width="239" RenderTransformOrigin="0.5,0.5" Content="Button">
  </BUTTON>
</BORDER>
```

This will create a red background behind the button.

**Placing and Customizing Visual Elements**

The visual elements available are defined in the XAML specification, and you will learn about each of them in detail in Chapter 4. Right now, let’s take a look at the basic shapes and tools that are available on the toolbar. These include the following shapes:

- **Rectangle** Select this shape to draw a straight-sided quadrilateral with 90-degree angles at each corner. You can make a square by creating a *Rectangle* with equal width and height properties.

- **Ellipse** Use this shape to draw an elliptical figure, an oval. You can make it a *Circle* by making the width and height properties equal.

- **Line** This shape simply draws a straight line between two end points.

There are also tools available on the toolbar that you can use to create free-form shapes:

- **Pen** Use this tool to draw a set of connected line segments represented by an underlying *Path* element.

- **Pencil** Use this tool to draw a set of connected elements, which can be lines or curves. Blend will take the strokes that the user draws and represent them with an underlying *Path* element.

Each of these visual elements, including those created with the Pen and Pencil tools, are represented by a single element, and this element can then be treated as any other object; that is, you can modify it in many ways, including setting its properties or animating it. For
example, consider Figure 2-19, in which the Pencil tool has been used to draw a set of connected curves to create a representation of the word *Hello* in script. Look on the Objects And Timeline view, and you’ll see the object represented as a *Path*.

![Figure 2-19 Editing a *Path* object.](image)

Now, this pencil “drawing” of the word *Hello* is treated as a single object, so you can edit its properties, including *Fill*, *Brush*, and so forth, simply by selecting it from the Objects And Timeline view and then editing the properties in the property pane, as with any other object.

### Placing and Customizing Controls

Controls are treated by Blend in exactly the same way as visual elements. You simply select them from the toolbar and draw them on the design surface. After you’ve created them on the design surface, then you can edit their properties. Controls are discussed in detail in Chapter 7, “Silverlight Controls: Presentation and Layout.”

One thing to note is that Blend gives you two families of controls on the toolbar. The first includes the Text controls: *TextBlock* and *TextBox*. The second includes the set of basic user interface controls: *Button*, *CheckBox*, *ListBox*, *RadioButton*, *ScrollBar*, *Slider*, and *GridSplitter*.

Finally, the toolbar gives you the option to add controls that aren’t part of this set. You can do this by selecting the Asset Library link at the bottom of the toolbar. This will display the Asset Library dialog box, as shown in Figure 2-20.
You can select controls in the Asset Library dialog box to add them to the toolbar. You also can search for specific controls by entering the term in the search box. So, for example, if you want to use a MediaElement control, start typing the letters of the control’s name. When you see the control you want (in this case, the MediaElement), you can select it, and it will then be available to you on the toolbar.

Then you can draw the control on the design surface and manipulate its properties with the properties editor, as you have done with the visual elements and layout controls.

Using Blend to Design Animations

We will examine how to create animations in detail in Chapter 5, but to put it succinctly, animations occur in Silverlight whenever a property of an object changes its value over time. You can design these kinds of animations visually in a very straightforward manner by using Blend and the timeline editor.

One form of animation that Silverlight supports is the DoubleAnimation, which is used to change numeric properties, such as the width of an Ellipse visual element. Another is the ColorAnimation, which is used to change the color of the Brush property.

For example, consider the Ellipse shown in Figure 2-21 (which appears as a circle because its height and width properties are equal). To visually design an animation that changes the width of this Ellipse element, you’ll add a new Storyboard that contains the animation.
On the Objects And Timeline view, select the *Ellipse* and then press the + button next to the *Storyboard* list at the top of the pane. Accept the defaults in the Create Storyboard dialog box that display, and then the Timeline editor will appear. You’ll also see the message Timeline Recording Is On at the top of the Blend window. Press the F6 key to rearrange the workspace so that the timeline is displayed to make it easier to work on an animation. Your screen should look something like the one shown in Figure 2-22.

Look for the yellow line in the timeline view. This denotes the *current* position on the timeline. Drag it to the 2-second mark, and then click the Record Keyframe tool at the top of the timeline. It looks like a blob with a little green plus sign (+) at its lower-right side. You’ll see a little oval that appears in the timeline at the 2-second mark, as shown in Figure 2-23.

Now that you have defined a keyframe, any changes that you make to the properties of the object will be recorded at that key frame, so go ahead and change the width of the *Ellipse* while the yellow line is still at the 2-second mark, indicating the current position of the timeline. For example, change the width to **200** and the *Fill* color to **Red**.

Now drag the playhead (the top of the vertical yellow line on the timeline) left and right, and you’ll see a preview of the animation previewed, with the width and color of the circle shape changing over time.
FIGURE 2-22 Editing the timeline.

FIGURE 2-23 Adding a key frame.
Introducing Silverlight 2

You can see the XAML that is generated by your visual creation of the animation here:

```xml
<UserControl.Resources>
  <Storyboard x:Name="Storyboard1">
    <DoubleAnimationUsingKeyFrames Storyboard.TargetName="ellipse"
      Storyboard.TargetProperty="(FrameworkElement.Width)"
      BeginTime="00:00:00">
      <SplineDoubleKeyFrame KeyTime="00:00:02" Value="200"/>
    </DoubleAnimationUsingKeyFrames>
    <ColorAnimationUsingKeyFrames Storyboard.TargetName="ellipse"
      Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
      BeginTime="00:00:00">
      <SplineColorKeyFrame KeyTime="00:00:02" Value="#FFFF2200"/>
    </ColorAnimationUsingKeyFrames>
  </Storyboard>
</UserControl.Resources>

<Ellipse Height="100" Width="100" Fill="#FFFFF500" Stroke="#FF000000" x:Name="ellipse"/>
```

You’ll delve into the structure of this XAML in much more detail in Chapter 5, but the important elements to note here are the `Storyboard.TargetName` instances that indicate which element the animation is being defined for, and the `Storyboard.TargetProperty` that indicates the property that is going to be changed. As you can see in this XAML, there are two animations, one that changes the width of the target and the other that changes its color. Silverlight then takes this definition and uses it to calculate the values required for each frame at the time the animation is rendered.

**Summary**

In this chapter, you learned the basics of working with Expression Blend, taking a quick tour of what it offers you as a designer or developer creating and implementing your own Silverlight applications. You saw how Blend can be used to create Silverlight solutions and projects, and then you saw what tools the Blend IDE offers you to add and manage visual elements, layout, controls, and animations in your application.

We’ve just begun to investigate what you can do with Blend in this chapter, but your introduction to Blend here may well inspire you to want to learn more about it.

The other half of the designer/developer workflow tool package is found in Visual Studio. In Chapter 3, you’ll take a look at how you can use this tool, what it has in common with Blend, and what powerful features it provides for developers. You will have the chance to use Visual Studio to build your first Silverlight application—a sliding picture puzzle game.
Chapter 3

Using Visual Studio with Silverlight 2

In Chapter 1, “Introducing Silverlight 2,” you were introduced to Silverlight 2 and saw the architecture that allows Silverlight and the .NET Framework to work together. You saw how XAML is used as a model for representing your user interface (UI) elements, interactions, and animations. In addition, you saw how the programming model for Silverlight can be hosted within the browser and programmed with JavaScript, or it can be hosted within a .NET Run-time for the browser and programmed using C# or other .NET languages.

Then, in Chapter 2, “Using Expression Blend with Silverlight 2,” you took a closer look at Expression Blend, learning more about the design tool that is used to build Silverlight experiences.

In this chapter, you’ll be taking a developer’s view of this process, getting some hands-on experience as you use Microsoft Visual Studio 2008 to build a simple sliding picture puzzle game. You’ll build on this example in later chapters. By the end of this chapter, you’ll have a good understanding of how to use C# and Visual Studio 2008 to build .NET-based applications.

Installing the Visual Studio Tools for Silverlight

The Visual Studio tools for Silverlight include a “chained” installer that takes care of installing the runtime (for Windows), the Silverlight 2 Software Development Kit (SDK), and the Visual Studio tools themselves. It can be downloaded from http://www.silverlight.net.

Please note that there are some prerequisites for installing the tools:

- The Web Authoring feature of Visual Studio must be installed.
- You must uninstall any previous versions of the Silverlight runtime before continuing.
- You must uninstall any previous versions of the Silverlight SDK before continuing.
- You must uninstall any previous versions of the Visual Studio Tools for Silverlight before continuing.
- Uninstall the Visual Studio update KB947520 before continuing.

If you haven’t met these criteria, the installer will fail, and you will see the dialog box shown in Figure 3-1.
FIGURE 3-1 Failed installation of the Visual Studio Tools.

If you have the correct prerequisites, you’ll see the dialog box shown in Figure 3-2.

FIGURE 3-2 Dialog box indicating that you are ready to install Visual Studio Tools for Silverlight.
Click Next, and you’ll see a screen where you can read and accept the license agreement. After you do so, the installation will begin immediately. You can see what the installation process should look like in Figure 3-3.

![Silverlight Tools Installation Wizard](image)

**FIGURE 3-3** Installing the Silverlight Tools.

When the installer finishes downloading and installing, you’ll have the Silverlight runtime, the SDK, and the Visual Studio tools all ready to go. Note that early versions of the SDK do not integrate the Help feature directly into Visual Studio. However, there is a Readme file located at `\Program Files\Microsoft SDKs\Silverlight\v2.0\Documentation\VS-Help` that will give you instructions on how to integrate it manually.

**Using Visual Studio to Build a Silverlight Application**

Now that you have installed the Visual Studio Tools for Silverlight, you are ready to learn how to use them to design and build an application. Figure 3-4 shows the Silverlight sliding picture puzzle game in action. This application has been written entirely in C# and XAML and is hosted in the browser.
In the following sections, you’ll see how to use Visual Studio 2008 and Silverlight to build this application in the C# language.

**Creating a Silverlight Application in Visual Studio 2008**

After you’ve installed Visual Studio and all of the necessary tools and templates for Silverlight, you will be able to create Silverlight applications. To do this, select New Project from the File menu. This will open the New Project dialog box (see Figure 3-5).

![Visual Studio 2008 New Project dialog box.](image)
Make sure that the .NET Framework 3.5 filter is selected from the drop-down list at the upper right in this dialog box (see Figure 3-5), and select Silverlight from the Project Type list. You’ll see that the Silverlight Application and Silverlight Class Library templates are available.

Select the Silverlight Application template type, and give your project a name and location. Click OK, and Visual Studio will launch the Silverlight Application Wizard (see Figure 3-6).

This wizard gives you several options for how you can create and manage your Silverlight application. All Silverlight applications in 2.0 are built as user controls that can then be instantiated and hosted on a page. Thus, this discussion concentrates on how you will do this.

The available options are:

- **Add A New Web To The Solution For Hosting The Control**  This option is the default option, and it will create a new Web site or Web application project that is configured to host and run your Silverlight application. It’s a useful shortcut for building new Silverlight applications (as you are doing in this book!) because it handles the coding details required to host the Silverlight control on a page, leaving you to focus on Silverlight itself. Use the options pane to select the type of project for the Web (it can be Web site or Web Application) and to name it.

- **Generate An HTML Test Page To Host Silverlight Within This Project**  This will create a new page at run time each time you try to debug and test your application. This is particularly useful if you want to concentrate solely on the Silverlight application and don’t want to worry about the overhead of having a separate Web project. Your final control can then be deployed to a server and instantiated on a Web page.

- **Link This Silverlight Control Into An Existing Web Site**  This option will allow you to link the Silverlight control to an existing Web site in your solution. If you are creating a
new project, the option will not be available. If you are adding a Silverlight application to an existing solution that has a Web project, you will be able to select this option.

Accept the defaults, as shown in Figure 3-6, and a new Visual Studio solution, containing a Silverlight control and a Web Site that hosts the control, will be created for you. You can see the project structure for this in Figure 3-7.

As you can see in Figure 3-7, two solutions have been created for you: the basic Silverlight control (*SlidingBlocks*) and a Web site. In the next section, you will examine these projects and learn what each one contains. Following that, you’ll start building the application.

### The Silverlight Control Project

The basic project that is created for you by the template contains a number of files, including the application manifest, the application XAML file with its code-behind, a sample page with its code-behind, the assembly information file, and some references. We’ll look at each of these files in turn. This section introduces some of the complexities of a Silverlight project,
which you might want to skip over if you just want to start coding, but I would recommend that you come back and read this information so that you can understand how everything hangs together.

**Understanding the Silverlight Project Properties**

The best way to get started is to look at the project properties. To do this, right-click the SlidingBlocks project in the solution explorer and select Properties. The Project Properties dialog box will appear (see Figure 3-8).

![Silverlight Project Properties dialog box.](image)

If you are familiar with this dialog box, you'll notice that there is an extra tab for Silverlight. This is selected in Figure 3-8, showing the Silverlight options.

The Assembly Name defaults to the project name. When the application is compiled into a DLL, this is the name that will be used.

The Default Namespace also defaults to the project name. If you reference classes within the project, they will be prefixed by this namespace.

The Startup Object defaults to the name of the project followed by .App (i.e., SlidingBlocks.App). This is a class in your application that will execute first. The template defines this class in App.xaml and its associated code-behind App.xaml.cs that you'll be looking at in a later section.

Clicking the Assembly Information button will call up the Assembly Information dialog box (see Figure 3-9). This allows you to define the metadata for your assembly, including Title, Description, Copyright, and Trademark information. This is stored in the AssemblyInfo.cs file and compiled into your Silverlight application.
Part I  Introducing Silverlight 2

The Package Outputs Into XAP File option allows you to specify an .xap file (new to Silverlight 2) that contains all of the outputs of your project compilation. Typically this is the DLL containing your Silverlight component and its application manifest. This is convenient for Web-based applications, as the Silverlight component only needs to be pointed at one file (the XAP file) to download and use multiple components.

![Assembly Information](image)

**FIGURE 3-9** Defining the Assembly Information.

**Tip**  The XAP file is just a ZIP file with a different extension. If you want to investigate its contents, just rename it with a .zip extension and open it with your favorite ZIP utility. Don’t forget to change the file extension back to .xap when you’re finished!

Finally, you are given the option to generate the Silverlight manifest file itself. This file contains details on everything within the package that the Silverlight application will use, such as additional components or controls that are necessary for your application to execute. You’ll be seeing more of this as you progress through this book.

**The Properties Files**

The first folder within your project is the Properties folder that contains the properties files: AppManifest.xml and Assemblyinfo.cs.

AppManifest.xml is generated for you as you compile your project. If your project has any dependencies at run time, such as external controls, references to them are placed here.

Assemblyinfo.cs contains the metadata to be compiled into your DLL that was configured in the Assembly Information dialog box (see Figure 3-9). You can manually change the information by editing this code file if you wish, but the recommended approach is to use the dialog box.
References
The References folder contains references to a number of assemblies. These are the core Silverlight assemblies that are needed to make your application run:

- **mscorlib** This contains the basic core types that are used by Silverlight applications.
- **System** This contains many of the high-level types used for developing and debugging Silverlight applications, such as the compiler and the debugging and diagnostics classes.
- **System.Core** This contains the LINQ data functionality.
- **System.Xml** This contains the Silverlight XML processing libraries.
- **System.Windows** This contains the core Windows and Silverlight functionality, including the Silverlight controls.
- **System.Windows.Browser** This contains the libraries used for interacting with the browser.
- **System.Windows.Controls** This contains the core set of Silverlight controls. It includes all the basic UI elements.
- **System.Windows.Controls.Extended** This contains the extended set of Silverlight controls. It includes complex additional controls such as the Calendar and WatermarkedTextBox controls.

Silverlight also has a number of nondefault assemblies that can be added to provide plug-in functionality, some of which you’ll be looking at through the course of this book. An example of this is the Dynamic Language Runtime functionality.

The App.xaml and App.xaml.cs Files
App.xaml is created for you by the integrated development environment (IDE) when you create a Silverlight project using the template. It is generally used to store application-global information.

App.xaml contains the declarations for the application’s behavior. Here’s an example of the default App.xaml that is created by the template:

```xml
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="SlidingBlocks.App">
  <Application.Resources>
  </Application.Resources>
</Application>
```

The first thing to take note of is the `x:Class` attribute, which specifies the name of the class into which this XAML and its associated code-behind will be compiled. As you can see, in this case it is SlidingBlocks.App, which you may remember from the Project Properties dialog box.
Part I Introducing Silverlight 2

back in Figure 3-8 is defined as the startup object for this application. Thus, when the Silverlight project is run, this class will contain the startup functionality.

You can specify a function to execute upon application startup using the \textit{Startup} attribute. This simply contains the name of the code function within the code-behind that will execute when the application starts. You can also specify the function to execute when the application finishes using the \textit{Exit} attribute, which contains the name of the function within the code-behind that will execute when the application closes.

The code for the default code-behind that is generated by the template is shown here:

```csharp
using System.Windows;
using System;

namespace SlidingBlocks
{
    public partial class App : Application
    {
        public App()
        {
            this.Startup += this.Application_Startup;
            this.Exit += this.Application_Exit;
            
            this.UnhandledException += this.Application_UnhandledException;
            InitializeComponent();
        }

        private void Application_Startup(object sender, EventArgs e)
        {
            // Load the main control
            this.RootVisual = new Page();
        }

        private void Application_Exit(object sender, EventArgs e)
        {
        }

        private void Application_UnhandledException(object sender, ApplicationUnhandledExceptionEventArgs e)
        {
        }
    }
}
```

First take a look at the constructor (which is the function with the same name as the code module, in this case \textit{App()}). It uses the code method to wire up the \textit{Startup} and \textit{Exit} functions. This was already done in the XAML file, so it isn’t necessary to do this in code—but it does show some of the nice flexibility of the XAML/code-behind model that allows you to wire up
events at design time (by specifying them in the XAML) or at run time (by declaring them in code).

Next, you can inspect the Application_Startup and Application_Exit functions. Notice that they take two parameters—the object that raised the event and an arguments object. You’ll be seeing this function signature often as you program your Silverlight applications.

The Application_Startup function contains code that sets the RootVisual property of the application to a new Page object, declaring that the UI of the Page object is the first UI screen that this application should render. If you are going to use other UI screens declared in XAML, these will be launched from within the Page object.

The Page object is the default XAML object that is created to host your application UI by the template, which you’ll be looking at in the next section.

**The Page.xaml and Page.xaml.cs Files**

The Page.xaml file provides the default UI for your application. When compiled along with its associated code-behind, it will form the Page class, from which a Page object can be created. If you recall from the previous section, the RootVisual property of the application was set to a new instance of a Page object, thus allowing this class to provide the default UI.

You can see the default XAML for Page.xaml here:

```xml
<UserControl x:Class="SlidingBlocks.Page"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400"
    Height="300">
    <Grid x:Name="LayoutRoot" Background="White">
    </Grid>
</UserControl>
```

First, you’ll notice that the container for the XAML is a UserControl, which is something you may not be familiar with from Silverlight 1.0. As mentioned earlier, when building Silverlight 2 applications in Visual Studio, you actually build controls that compile into a DLL within an XAP that Silverlight opens and renders.

In this case, you’ll see that this UserControl instance is called SlidingBlocks.Page. SlidingBlocks is the namespace (take a look back at the project properties to see this), and Page is the name of the class within that namespace.

The xmlns and xmlns:x declarations configure the default namespace and the extended namespace, respectively, to be used to validate the XAML. Earlier you saw the x:Class attribute
used to define the class for this control, and this is an example of using the extended namespace, which is prefixed by x:

Finally, the width and height are set to the default 640 × 480.

Next comes the root Grid. In Silverlight 2, your root element must be a Container, which in this case is a Grid called LayoutRoot. All elements of your UI design will ultimately be children of this node.

The code-behind for this XAML is shown here:

```csharp
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace SlidingBlocks
{
    public partial class Page : UserControl
    {
        public Page()
        {
            // Required to initialize variables
            InitializeComponent();
        }
    }
}
```

If you are familiar with C#, this will look very similar to code you may have used before. Basically, it is a boilerplate class file named Page that inherits from the UserControl type. In the class constructor, the special InitializeComponent() call is used to set everything up. You add your page-specific code to this module, as you'll see in the section titled "Building a Silverlight 2 Game" later in this chapter, where you will create the sliding picture puzzle.

**The Web Project**

In addition to the control project, the template also created a Web project that hosts your Silverlight application. This Web project contains two ASPX files: Default.aspx, which is an empty Web Form on which you can build an application; and a test page called <ApplicationName>TestPage.aspx (e.g., SlidingBlocksTestPage.aspx), which contains everything necessary to run Silverlight from ASP.NET.
Although Silverlight does not have any server-side dependencies, ASP.NET offers some controls that allow the generation of the client-side JavaScript and HTML necessary to host Silverlight in the browser.

The TestPage file includes references to these controls. Following is the full markup for the ASPX file:

```xml
<%@ Page Language="C#" AutoEventWireup="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title>Test Page For SlidingBlocks</title>
</head>
<body style="height:100%;margin:0;">
<form id="form1" runat="server" style="height:100%;">
<div>
<asp:ScriptManager ID="ScriptManager1" runat="server">
</asp:ScriptManager>
<asp:Silverlight ID="Xaml1" runat="server"
    Source="~/ClientBin/SlidingBlocks.xap"
    Version="2.0" Width="100%" Height="100%" />
</div>
</form>
</body>
</html>

Note that this is an evolving technology, so your version number attributes and public key attributes may differ slightly. Don’t worry—if your code was generated by the template, you should be in good shape.

You’ll notice that there are two ASP.NET controls referenced on this page. The first is the `ScriptManager` control, which is an artifact of ASP.NET AJAX and is a terrific control that is used to manage the downloading and referencing of all necessary JavaScript libraries at the correct time and in the correct place.

The second is the `Silverlight` control. Notice that it takes the XAP that we discussed earlier as its parameter. This control will generate the correct HTML code to create the `DIV` and `CreateSilverlight` functions.

When you run this page, you’ll see that a lot of HTML and JavaScript is generated. Toward the bottom of the code, shown in bold here, you’ll see where Silverlight is created and pointed at the XAP file. Here’s a snippet:
This script is interpreted by the browser to instantiate the Silverlight control. Please note that this code relies on ASP.NET and the ASP.NET Silverlight controls to work properly. If you are instantiating from something other than ASP.NET, you can still use the SDK-based JavaScript tools to instantiate Silverlight. This is covered in depth in Chapter 6, “The Silverlight Browser Control.”

Building a Silverlight 2 Game

Silverlight follows the model of separating the design from the development by having the design in XAML technology and the code for the development in code-behind technology, typically (though not exclusively) programmed with C#.

In this example, you’ll use very little XAML—just enough to contain the Canvas in which the puzzle pieces will be kept and the image that shows the completed image. If you refer back to Figure 3-4, you will see these on the left and right, respectively. Note that although the default page has a Grid as its main container, we’ll use a Canvas in this game for simplicity.

Creating the UI in XAML

As you saw in Figure 3-4, the UI for this game is very simple, consisting of an area of the screen (contained within a Canvas) where the sliding blocks reside and another area that renders the finished image.

Here’s an example of the XAML that provides the Canvas and the completed image:

```xml
Width="640"
Height="480">
    <Canvas x:Name="LayoutRoot">
        <Canvas x:Name="GameContainer"/>
        <Image Source="sl.JPG" Canvas.Left="500"
            Height="400" Width="400" Stretch="UniformToFill"/>
    </Canvas>
</UserControl>
```
As you can see, it is very straightforward. The Canvas that will contain the pieces of the puzzle is called GameContainer, and the image that renders the completed puzzle image is static and thus does not need to be named. It is prefilled with the image sl.jpg that exists within the same Web project. That’s everything you need for your design, so in the next section, you’ll start looking at the code.

Writing the Game Code

The game code in this example is written using C#. You could easily translate it into VB.NET, IronPython, or any other supported language that you want to use, but in most cases, this book will use C#.

Initializing the Data Structures

The first step in writing the code for this game is in initializing the data structures that will be used by the application. In this case, we are going to break the image up into 16 tiles, 15 of which will be shuffled across the board. One tile is not used; instead, there is a blank space that is used by the player to slide the blocks around.

In Silverlight, you cannot create a subimage from an existing image, but you can clip an image according to a clipping path. Clipping shouldn’t be confused with cropping. In the former, you dictate which parts of the image to draw; in the latter, you remove all but the desired part of the image. Silverlight does not support cropping, so you have to clip an image.

The problem with clipping is that the rest of the image dimensions are still available and clickable. So, for example, if you have a 400 × 400 image and you clip a 100 × 100 square at position (100,100), you will still have a 400 × 400 object that will be blank (but still clickable) except for a 100 × 100 square at position (100,100). This doesn’t suit our needs to create a sliding picture puzzle, so what can we do?

The answer comes by containing the image within a Canvas that is the same dimensions as the clip and then using a translate transform on it so that the clipped part of the image is at the upper-left corner of this Canvas, and thus the Canvas only renders the clipped region.

So, for a 4 × 4 puzzle made up of 16 blocks, you need 16 images and 16 Canvas objects. You’ll also need something to represent the board so that you know which image is in which square. Here’s the code to declare these:

```csharp
Canvas[] cI = new Canvas[16];
Image[] i = new Image[16];
int[] board = new int[16];
```
Creating the Puzzle Pieces

To create the puzzle pieces, you will have to load the sl.jpg image into each element in the array of images. This is achieved by using a uniform resource identifier (URI) to point to the image and the `BitmapImage` class (from `System.Windows.Media.Imaging`) to read the data from that URI and point it at the image control:

```csharp
i[nx].Source = new BitmapImage(uri);
```

From your constructor, you can call a function (`InitBoard`) that will be used to instantiate the pieces:

```csharp
public Page()
{
    // Required to initialize variables
    InitializeComponent();
    InitBoard();
}
```

You can see this function here:

```csharp
void InitBoard()
{
    Uri uri = new Uri("sl.jpg", UriKind.Relative);
    int nx = 0;
    for (int ix = 0; ix < 4; ix++)
        for (int iy = 0; iy < 4; iy++)
        {
            nx = (ix * 4) + iy;
            i[nx] = new Image();
            i[nx].Height = 400;
            i[nx].Width = 400;
            i[nx].Stretch = Stretch.UniformToFill;
            RectangleGeometry r = new RectangleGeometry();
            r.Rect = new Rect((ix * 100), (iy * 100), 100, 100);
            i[nx].Clip = r;
            i[nx].Source = new BitmapImage(uri);
            i[nx].SetValue(Canvas.TopProperty, Convert.toDouble(iy * 100 * -1));
            i[nx].SetValue(Canvas.LeftProperty, Convert.toDouble(ix * 100 * -1));
            cI[nx] = new Canvas();
            cI[nx].Width = 100;
            cI[nx].Height = 100;
            cI[nx].Children.Add(i[nx]);
            cI[nx].SetValue(Canvas.NameProperty, "C" + nx.ToString());
            cI[nx].MouseLeftButtonDown += new
            MouseButtonEventHandler(Page_MouseLeftButtonDown);
            if (nx < 15)
                GameContainer.Children.Add(cI[nx]);
        }

    // Mix up the pieces
    shuffle();
}
// Draw the board
drawBoard();
}

Be sure to add a reference to System.Windows.Media.Imaging at the top of your code page:


At first, this may look a little complex, but on closer inspection, it is actually quite straightforward. A nested loop from 0–3 on the x- and y-axes is set up. This, as you may have guessed, is used to manage the $4 \times 4$ array for the images themselves.

The Image and Canvas arrays used to store the blocks are one-dimensional arrays with 16 elements each (as less memory and code are used to store them this way), so to figure out how to map a two-dimensional $ix, iy$ coordinate to a one-dimensional array, the following calculation is needed:

$$nx = (ix \times 4) + iy;$$

Each Image element is then set up with a $400 \times 400$ dimension with UniformToFill stretch. You can find more about how to use images in Chapter 4, "XAML Basics."

Next, the clip region is calculated. This is done using a RectangleGeometry:

```csharp
RectangleGeometry r = new RectangleGeometry();
r.Rect = new Rect((ix*100), (iy*100), 100, 100);
i[nx].Clip = r;
```

This defines a Rectangle at the appropriate coordinates (derived by multiplying $ix$ and $iy$ by 100) with the appropriate size ($100 \times 100$) and then assigns it to be the clipping region for the current image ($i[nx]$).

Next, the image is loaded by setting it to the value of the BitmapImage, which is initialized from the URI of the image, and it is translated into position. An image that is clipped at position ($100,100$) has to be moved by ($-100,-100$) for the clipped area to appear in the upper-left corner. Thus, the Top and Left properties have to be set to $-100$ multiplied by the current $iy$ and $ix$ values, respectively.

```csharp
i[nx].Source = new BitmapImage(uri);
i[nx].SetValue(Canvas.TopProperty, Convert.ToDouble(iy * 100 * -1));
i[nx].SetValue(Canvas.LeftProperty, Convert.ToDouble(ix * 100 * -1));
```

The images have not yet been added to a parent Canvas, so this is the next step. The Canvas needs to be initialized, sized, and the respective image added to it as a child. Here's the code:

```csharp
cI[nx] = new Canvas();
cI[nx].Width = 100;
```
Part I  Introducing Silverlight 2

Now that we have our Canvas and it contains our clipped image, we complete our initialization by naming our Canvas (so that we can track it later when we click it), defining an event handler to manage what happens when the user clicks it, and finally adding it to the parent Canvas. We don’t need to position the block yet. That will happen after the board is shuffled. Note that we don’t add the final image to the board because we want to have an empty space.

cI[nx].SetValue(Canvas.NameProperty, "C" + nx.ToString());
cI[nx].MouseLeftButtonDown += new MouseButtonEventHandler(Page_MouseLeftButtonDown);
if(nx<15)
GameContainer.Children.Add(cI[nx]);

Finally, we want to shuffle the pieces and draw the game board. You’ll see the code for this in the next section.

Shuffling the Pieces

The puzzle pieces are shuffled using a fairly simple shuffle algorithm that goes through the array 100 times, picking out two random elements on each occasion; if the two elements are different, it swaps their contents. At the end, it loads the value -1 into the last element to recognize that it is the empty square.

You can see the shuffle algorithm here:

```csharp
void shuffle()
{
    // Initialize Board
    for (int n = 0; n < 15; n++)
    {
        board[n] = n;
    }
    Random rand = new Random(System.DateTime.Now.Second);
    for (int n = 0; n < 100; n++)
    {
        int n1 = rand.Next(15);
        int n2 = rand.Next(15);
        if (n1 != n2)
        {
            int tmp = board[n1];
            board[n1] = board[n2];
            board[n2] = tmp;
        }
    }
    board[15] = -1;
}
```
Now that the pieces are shuffled, our next step will be to draw the board.

**Drawing the Board**

At this point, you have all of the blocks defined as *Image* elements within *Canvas* elements, and you have an array of integers, where the value at index \( n \) is going to be the tile to display at that position. You’ve also shuffled this array of integers, so now it’s time to draw the game board. This is achieved simply by using this code:

```csharp
void drawBoard()
{
    int nx = 0;
    int ny = 0;
    for (int n = 0; n < 15; n++)
    {
        nx = n / 4;
        ny = n % 4;
        if(board[n]>=0)
        {
            cI[board[n]].SetValue(Canvas.TopProperty, Convert.ToDouble(ny * 100));
            cI[board[n]].SetValue(Canvas.LeftProperty, Convert.ToDouble(nx * 100));
        }
    }
}
```

This will loop from 0 to 14 (there are 15 blocks in the puzzle) and calculate an \( x,y \) coordinate for each block in a \( 4 \times 4 \) grid. The \( x \) value is simply the integer division of the loop index by 4, and the \( y \) value is simply the modulus of the loop index by 4. If we multiply these by 100, we then get the right position to draw the *Canvas* element.

At this point, we have fully initialized the game. The image is positioned on the right, and our shuffled board of image blocks is on the left.

**Handling User Control**

The next thing to do is to start handling the user interaction. In a game such as this, the expected behavior is that the user clicks an image block, and if this block is next to the empty space, the block that the user clicked will slide into the space, leaving a new empty space behind. So, we need to handle the clicking of the *Canvas* containing the Image block. If you remember all the way back to the initialization of the blocks, you saw this line:

```csharp
cI[nx].MouseLeftButtonDown
    += new MouseButtonEventHandler(Page_MouseLeftButtonDown);
```

This defined that the *Page_MouseLeftButtonDown* event handler will fire when the *Canvas* is clicked. This event handler has been wired up for each of the *Canvas* blocks.
The code for this event handler has two sections. The first section identifies which Canvas raised the event and where that Canvas is in the board:

```csharp
void Page_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Canvas c = sender as Canvas;
    int nCanvasID = -1;
    int nBoardLoc = -1;
    int nEmptyLoc = -1;
    for (int i = 0; i < 16; i++)
    {
        if (c == cI[i])
        {
            nCanvasID = i;
            break;
        }
    }
    for (int i = 0; i < 16; i++)
    {
        if (board[i] == nCanvasID)
        {
            nBoardLoc = i;
        }
        else if (board[i] == -1)
        {
            nEmptyLoc = i;
        }
    }
}
```

So, for example, you may have clicked on the block that represents the upper-left corner of the finished image (block 0), but it is currently in the lower-left corner of the board (position 12). When the click event is raised, you would look through the array of Canvas elements that represent the blocks until you find one that matches the Canvas that was actually clicked, and from here you could get its index in the array, loading it into the nCanvasID variable (which in our previous hypothetical case would be 0). You can then scan through the board to find where item 0 is and, when you find it, assign this value to the nBoardLoc variable (which in our hypothetical case is 12) and, while you are at it, find the location of the empty space on the board and load that into nEmptyLoc.

The second section of code then needs to check to see if we can move, and if we can, it moves the block into the space and updates the board accordingly.

```csharp
    // Check if we can move
    if ((nBoardLoc == nEmptyLoc + 1) ||
        (nBoardLoc == nEmptyLoc - 1) ||
        (nBoardLoc == nEmptyLoc + 4) ||
        (nBoardLoc == nEmptyLoc - 4))
    {
        int nx = nEmptyLoc/4;
        int ny = nEmptyLoc%4;

        cI[nCanvasID].SetValue(
```
To do this, you first check the position of the empty location relative to the position of the location of the block that you clicked. If it is immediately above, below, to the left, or to the right of the current block, you can move it. Because the board is a one-dimensional array representing a 4 × 4 board, this is easy to do. Items to the left and to the right of the current item are off by −1 and +1, respectively, and items above and below are off by −4 and +4, respectively, so if you seek the blocks at these indices for the empty block, you know that you can move.

To move, you then need to get the x and y coordinates relative to the position of the empty block and assign them to the position of the clicked block. Then assign the board location that previously contained the empty block to contain the value of the Canvas that you just clicked, and the board location that previously contained the Canvas that you just clicked to −1, indicating that it is now empty.

Finally, you’ll call the checkWinner() function to see if the board has been successfully unscrambled.

### Checking Winning Condition

The checkWinner function checks to see if the board has been successfully unscrambled. This, again, is very straightforward. The board is unscrambled if every item at index n in the board array is equal to n—that is, Canvas 0 is at index 0, Canvas 1 is at index 1, and so on.

The most efficient way to do this is to assume that you have a winning board and scan through it until board[n] is not equal to n, at which point you have a losing board and you break the loop. If you make it all the way through the board, you have a winner.

```csharp
void checkWinner()
{
    bool bCompleted = true;
    for (int n = 0; n < 15; n++)
```
Part I Introducing Silverlight 2

{  
    if (n != board[n])  
    {  
        bCompleted = false;  
        break;  
    }  
}  
if (bCompleted)  
{  
    // The Player has won the game....do something nice for them.  
}  
}

At this point, you have all the elements for a basic image sliding picture puzzle game written in C# for Silverlight 2, and it was done in just over 100 lines of code. You can build on this sample as you progress through this book—adding animation to the sliding of the blocks, saving high scores, allowing images to be uploaded to the application, and so forth. The sky is the limit to the enhancements you can add!

Summary

In this chapter, you took a look at Visual Studio 2008 and the various tools and templates that it offers you to develop Silverlight applications using .NET languages. You took a tour of a project based on the default Silverlight templates, inspecting each of the files and how it is used to develop and deploy a Silverlight application. You then put the theory into practice by using XAML and C# to build a fully featured sliding picture puzzle game. This gives you a taste of what you can do with Silverlight 2, but it barely scratches the surface of what is possible. In Part 2 of this book, “Programming Silverlight 2,” you’ll start looking in more depth at some of the major areas of functionality that are available from the .NET Framework, including building your own controls, networking and communication, data and XML, dynamic languages, and the ASP.NET server controls.