In the first chapter, we talked about how the construction of a framework like WPF is much like the construction of a house. If you don’t know why certain things are built the way they are, you are likely to use them improperly and break something.

This chapter is all about the tools you use when building your house. Every craftsman (including programmers!) knows that picking the right tool for the job is essential to the success of the project. If you use a tool that has too much power, you’re likely to break something or punch a hole through a wall. Go with something that doesn’t have enough power and you won’t be able to get the job done either.

WPF provides a rich and diverse set of classes that allow you to create everything from simple visuals to complex layered visuals and components. This is possible because of the precision with which the class structure of WPF was built. There are dozens of tools, but it is up to you to pick the right one for the job. Each class has a specific purpose and unique strengths that separate it from other classes. This allows us to mix and match classes to fit our particular needs.

Figure 2.1 shows the visual hierarchy of classes that we examine in detail in this chapter.

**Introducing the Visual Classes**

WPF has a rich, diverse set of building blocks and tools that you can use to create amazing interfaces. Knowing which tool to use and when to use it is absolutely invaluable to
creating next-generation applications. What follows is a brief overview of the most important classes in WPF. These are the classes that you will use most often as you progress through this book and as you create your own applications.

The **DispatcherObject Class**

The DispatcherObject class can be found in the System.Windows.Threading namespace. It provides the basic messaging and threading capabilities for all WPF objects. The main property you will be concerned with on the DispatcherObject class is the Dispatcher property, which gives you access to the dispatcher the object is associated with. Just like its name implies, the dispatching system is responsible for listening to various kinds of messages and making sure that any object that needs to be notified of that message is notified on the UI thread. This class does not have any graphic representation but serves as a foundation for rest of the framework.

The **DependencyObject Class**

The DependencyObject class provides support for WPF's dependency property system. The main purpose behind the dependency property system is to compute property values. Additionally, it also provides notifications about changes in property values. The thing that separates the WPF dependency property system from standard properties is the
Introducing the Visual Classes

ability for dependency properties to be data bound to other properties and automatically recompute themselves when dependent properties change. This is done by maintaining a variety of metadata information and logic with the DependencyProperty.

DependencyObject also supports attached properties, which are covered in Chapter 6, “The Power of Attached Properties,” and property inheritance.

The DependencyObject class is part of the System.Windows namespace and has no graphic representation. It is a subclass of DispatcherObject.

The Visual and DrawingVisual Classes

The System.Windows.Media.Visual abstract class is the hub of all drawing-related activity in WPF. All WPF classes that have a visual aspect to their nature are descendants in some way from the Visual class. It provides basic screen services such as rendering, caching of the drawing instructions, transformations, clipping, and of course bounding box and hit-testing operations.

While the Visual class contains a tremendous amount of useful functionality, it isn’t until we get down to the DrawingVisual class in the hierarchy that we start seeing concrete implementations that we can work with. DrawingVisual inherits from ContainerVisual, a class that is designed to contain a collection of visual objects. This collection of child visuals is exposed through the Drawing property (of type DrawingGroup).

DrawingVisual is a lightweight class specifically designed to do raw rendering and doesn’t contain other high-level concepts such as layout, events, data binding, and so on. Keep in mind the golden rule of this chapter: Pick the right tool for the job. If you need to simply draw graphics and the extent of user interaction with that object is simple hit testing, you can save a lot on overhead by using DrawingVisual.

A great example of where DrawingVisuals would be an excellent choice is in a charting application. You can build a variety of charts by using the drawing primitives such as lines, beziers, arcs, and text and fill them with colors using a solid brush or even more advanced fills such as linear and radial gradients.

You might be wondering what to do for your charting application if you need the charts to be data bound. You see more about how to do this later, but remember that the output of processing a data template can be simple drawing visuals, allowing you to create data-bound charts that produce only the functionality you need.

Listing 2.1 shows an example of drawing a sector in a chart. In charting terms, a sector is a closed region that looks like a pizza slice. It has two straight lines that form the two sides of a triangle, but the last piece of the shape is closed by an arc rather than another straight line.

Listing 2.1 A “Pizza Slice” Sector Visual Class

```csharp
public class SectorVisual : DrawingVisual
{
    public SectorVisual
```


```csharp
{ 
    StreamGeometry geometry = new StreamGeometry;
    using (StreamGeometryContext c = geometry.Open)
    {
        c.BeginFigure(new Point(200, 200),
                      true /* isFilled */, true /* isClosed */);

        // First line
        c.LineTo(new Point(175, 50), true /* isFilled */, true /* isClosed */);

        // Bottom arc
        c.ArcTo(new Point(50, 150), new Size(1, 1), 0, true,
                SweepDirection.Counterclockwise, true /* isFilled */, true /* isClosed */);

        // Second line
        c.LineTo(new Point(200, 200), true /* isFilled */, true /* isClosed */);
    }

    // Draw the geometry
    using (DrawingContext context = RenderOpen)
    {
        Pen pen = new Pen(Brushes.Black, 1);
        context.DrawGeometry(Brushes.CornflowerBlue, pen, geometry);
    }
}
```

When rendered, the preceding class creates a visual that looks like the one shown in Figure 2.2.

If you have done any graphics programming for other platforms before, the concept behind the `DrawingContext` class should be pretty familiar to you. It is essentially an entry point into the conduit between your code and the actual rendered pixels on the user's monitor. As WPF is a retained graphics system, it caches all the drawing instructions and renders them whenever a refresh is required. The `DrawingContext` is used as the cache from which these instructions are picked up. In the preceding code, we start by building the geometry of the sector using the `StreamGeometryContext`. We then use the `DrawingVisual`'s `RenderOpen` method to obtain a reference to the current `DrawingContext` instance and draw the geometry. The `DrawingContext` class contains methods for drawing lines, rectangles, geometry, text, video, and much more. Using these methods, you can build up a shape like the sector in Listing 2.1.
While the `DrawingVisual` class is ideally suited to scenarios in which you just need to do basic drawing and hit testing, it still needs a container that is responsible for placing those graphics on the screen. One such container is the `FrameworkElement` class.

The `FrameworkElement` Class

`System.Windows.FrameworkElement` derives from `UIElement`, which actually provides the core services such as layout, eventing, and user input that are used by rest of the framework. Although `UIElement` is a public class you would typically not derive from it. Instead, the `FrameworkElement` makes a better choice since it exposes the previous services (that is, layout, styles, triggers, data binding) in a user-customizable way.

`FrameworkElement` is also a lightweight container host for a set of visuals. Because it is a descendant of `UIElement` it is free to participate in the logical tree and can provide container support for more primitive visual elements (such as the `DrawingVisual` from the preceding example). The `FrameworkElement` class can be used in the following ways:

1. Provide simple visual representations of data by overriding the `OnRender` method.
2. Compose custom visual trees, making the `FrameworkElement` an excellent container class.
3. Provide custom layout logic (sizing and positioning) for the contained visuals.
4. A combination of the above.

For the pie slice control to be displayed onscreen, we need to build a container in which the `SectorVisual` class (refer to Listing 2.1) is the lone visual child, as shown in Listing 2.2.

---

**NOTE**

**Retained Mode Graphics**

Remember that WPF is a retained-mode graphics system, which means all of the drawing instructions are cached and you do not need to call any kind of update graphics API to force a visual refresh, as in an immediate mode graphics system. Although the API around `DrawingVisual` and `DrawingContext` resembles something you find in an immediate mode graphics system, beware of using it like one. You should never have to call any kind of update-my-graphics API to force a visual to redraw.
LISTING 2.2 Creating a Container for the SectorVisual Class

```csharp
public class VisualContainer : FrameworkElement
{
    private SectorVisual _visual = new SectorVisual();

    protected override Visual GetVisualChild(int index)
    {
        return _visual;
    }

    protected override int VisualChildrenCount
    {
        Get
        {
            return 1;
        }
    }
}
```

It is worth pointing out that the preceding VisualContainer class could also have been a subclass of UIElement instead of FrameworkElement, since it is not doing any custom layout. A FrameworkElement is best suited when you also want to provide custom sizing and positioning of elements, data binding, and styles.

The Shape Class

The Shape class provides yet another mechanism to enable primitive drawing in WPF applications. If we already have the DrawingVisual, which we have seen can be used to draw lines, arcs, and “pie slice” wedges, what do we need the Shape class for?

The Shape class actually provides a level of abstraction slightly above that of the DrawingVisual. Rather than using the primitives of the DrawingContext as we have already seen, instead we can use the concept of geometry to determine what is going to be drawn.

As a developer creating a custom shape, you use the DefiningGeometry property on your custom shape class. This geometry defines the raw shape of the class, and other properties such as the stroke, stroke thickness, and fill determine the rest of the information needed to render the shape. If you have ever used shapes, strokes, and fills in Adobe Photoshop or Illustrator, these concepts should already be familiar to you. Whatever you create using DefiningGeometry can also be done using the more primitive DrawingVisual class, but...
using the geometry allows your custom shape class to be inserted more easily into a logical tree, making it more flexible and more amenable to reuse and packaging.

Shape is a subclass of FrameworkElement, a base class used by most container-type classes such as Panel to render child elements. This lets Shape instances participate in the layout pass and allows for easier event handling. Shape also defines the Stretch property, which allows you to control how a shape’s geometry is transformed when the dimensions of the Shape object change.

Figure 2.3 illustrates a sector shape and how it can be transformed automatically using the Stretch property.

![Figure 2.3](image)

**FIGURE 2.3** Stretching a shape’s *DefiningGeometry*.

Taking the previous example of the sector and upgrading it this time to inherit from the Shape class, we end up with the code in Listing 2.3.

**LISTING 2.3** Making the SectorVisual into a Shape

```csharp
public class SectorShape : Shape
{
    protected override Geometry DefiningGeometry
    {
        get { return GetSectorGeometry(); } 
    }

    private Geometry GetSectorGeometry()
    {
        StreamGeometry geometry = new StreamGeometry();
        using (StreamGeometryContext c = geometry.Open())
        {
            c.BeginFigure(new Point(200, 200), true, true);
            c.LineTo(new Point(175, 50), true, true);
        }
    }
}
```
As you can see from the preceding code, the construction of the shape is exactly the same as constructing a visual-based sector. The difference here is that for a Shape we stop after creating the geometry and setting that to the DefiningGeometry property. With the SectorVisual, we must both construct the geometry and render it. The core difference is basically a difference in responsibilities. The Shape knows how to render itself in its container using the geometry defined in DefiningGeometry.

When creating a shape's defining geometry, the most commonly used geometry classes are PathGeometry, StreamGeometry, GeometryGroup, or CombinedGeometry. You learn more about these types of geometry in more detailed examples later in the book.

**The Text Classes**

Developers often overlook fonts when they are digging in their toolbox for something to get the job done. WPF actually has robust support for drawing text, laying out text, and working with documents. Text can be displayed onscreen in multiple ways and ranges from simple text to text with complex layout and formatting support.

At the most primitive level, we have GlyphRuns and FormattedText. These can't be used declaratively; rather, you need to use the DrawingContext to display them onscreen. This can be done using the DrawingContext.DrawGlyphRun and DrawingContext.DrawText APIs.

In today's modern age of globalized applications, you need more than just the ability to blindly throw text onto the user interface. You need to be able to do things like display text that runs from right to left, display Unicode characters, and much more. For example, when you draw text into a drawing context, not only do you supply font information, but you also supply the text, text culture, flow direction, and the origin of the text:

```csharp
drawingContext.DrawText(
    new FormattedText("Hello WPF!",
    CultureInfo.GetCultureInfo("en-us"),
    FlowDirection.LeftToRight,
    new Typeface("Segoe UI"),
    36, Brushes.Black),
    new Point(10, 10));
```
Text can also be displayed declaratively and easily using the TextBlock and Label classes. TextBlocks (and Labels) are generally useful for a single line of text with fairly rich formatting and simple alignment support. For more complex text display, you can use the FlowDocument and FixedDocument classes that have more elaborate features to handle dynamic layouts, paragraphs, and mixing of rich media.

FlowDocument handles automatic layout and sizing of text and graphics as you resize the document. They are most useful for viewing newspaper-style text that can flow into columns and multiple pages. FixedDocuments are useful for programmatically generating a document with precise control over sizes of the textual elements, hence the name. These documents use two kinds of elements: blocks and inlines. Blocks are container elements that contain the more granular inline elements. Typical block-related classes include Paragraph, Section, List, and Table. Some of the common inline classes are Run, Span, Hyperlink, Bold, Italic, and Figure.

Although TextBlock, Label, FixedDocument, and FlowDocument are useful for displaying static text, WPF also provides interactive controls for editing text. These include the classic TextBox, which has limited formatting capabilities, and the RichTextBox, which as the name suggests has richer editing capabilities.

Most of these text-related classes expose properties to control alignment, fonts, font styles, and weights. Additionally, there is a class called Typography under the System.Windows.Documents namespace that has a rich set of properties to specifically control the various stylistic characteristics of OpenType fonts. They are available as attached properties, which can be set on text-related classes that use OpenType fonts. A sampling of the properties include Capitals, CapitalSpacing, Fraction, Kerning, and NumeralAlignment.

**The Control Class**

The Control class is pretty close to the top of the food chain of visual classes. It provides a powerful Template property (of type ControlTemplate) that can be used to change the entire look and feel of a control. Knowing that control templates can be changed during design time and at runtime can make for some amazingly powerful applications and compelling UIs. Designing with a Control allows developers and designers to quickly and easily define visual elements.

A rich set of classes that derive from the Control class provide specialized functionality and increasing complexity and level of abstraction. Choosing the right subclass of Control goes back to the analogy of choosing the right tool for the job. You need to make sure that you don't take something overly complex as well as not picking something that is too simplistic and doesn't offer the functionality you need. Choosing the wrong subclass can dramatically increase the amount of work you need to do.

For example, if you are building a control that needs to display a list of child items, you should start with ItemsControl or ListBox instead of starting with the comparatively low-level functionality of the Control class.
Unlike the earlier UI frameworks, the Control-related classes in WPF can be used directly without subclassing. Because of the powerful features such as Styles and Templates, you can customize the look and feel of a control declaratively. The subclasses of Control deal with the shape of the data rather than the appearance. A Button deals with singular data. ScrollBars, Sliders, and so on work with range data. ListBox and ListView work with collections. TreeView works with hierarchical data. It is up to the development team to decide how best to visually represent the data using these controls. In most cases, you do not have to subclass a control, rather you only have to change its Style and Template.

The ContentControl Class
The ContentControl class is ideal for displaying singular content, specified via the Content property. The content’s look and feel can be customized using its ContentTemplate property, which is of type DataTemplate. Remember back in Chapter 1, “The WPF Design Philosophy,” how plain data gets transformed into a visual representation through data templates.

The container that hosts the content can also be customized using the Template property of type ControlTemplate. This way you actually have two levels of customization available to you: You can customize the outer containing frame (via the Template property), and you can customize how the content within the frame is rendered (via the ContentTemplate property).

Controls derived from ContentControl are used to represent individual items that are displayed within list-based controls such as a ListBox, ItemsControl, ListView, and so on. The Template property is used for user interaction features such as showing selections, rollovers, highlights, and more. The ContentTemplate property is used for visually representing the data item associated with the individual element.

For example, if you have a list of business model objects of type Customer that you are displaying inside a ListBox, you can use its ItemTemplate property (of type DataTemplate) to define a visual tree that contains the customer’s picture, home address, telephone number, and other information. Optionally you can also customize the item container holding each Customer object. As mentioned, a ContentControl derived class is used for wrapping each item of a ListBox. We can customize this ContentControl derived container using its Template property, which is of type ControlTemplate.

Some of the most powerful tricks in WPF revolve around control templates, content controls, and content presenters, so it is well worth the effort of learning them in detail.

The ContentPresenter Class
The ContentPresenter class is the catalyst that brings a data template to life. It is the container that holds the visual tree of the data template. ContentPresenters are used inside the ControlTemplates of Control, ContentControl, or any other custom control that exposes a property of type DataTemplate. It may help to think of the role of the ContentPresenter as the class that is responsible for presenting the visual tree of a data template within its container.
Within the ControlTemplate, you associate the DataTemplate property of the template control with the ContentTemplate property of the ContentPresenter. You might do this in XAML (eXtensible Application Markup Language) this way:

```xml
<ContentPresenter ContentTemplate={TemplateBinding ContentTemplate} /> 
```

In the preceding snippet, we are template binding the ContentTemplate property of the ContentPresenter to the ContentControl’s ContentTemplate property.

In general, you can think of a presenter element as a shell or container for the actual content. It instantiates the template tree and applies the content to it. As you may recall from Chapter 1, you can think of the content as being a piece of cookie dough, the template is the cookie cutter, and the presenter pushes down on the dough and presents the end result of a nicely shaped cookie.

**The ItemsControl Class**

As this class’s name suggests, the ItemsControl class is ideally suited to displaying a list of items. More specifically, those items are interactive controls.

Not so long ago, when the main framework for building Windows applications was Windows Forms using .NET, controls were almost always too specialized. A ComboBox would display a drop-down list of items, but those items were always text, unless you rolled up your sleeves and did some serious work. This same problem occurred in virtually every place where Windows Forms displayed a list of items—the type and display of each item in a list was fixed unless you practically rewrote the control.

With WPF, the ItemsControl allows you to present a list of items that can have any visual representation you choose and can be bound to any list-based data you want. Finally we have both the flexibility we have always wanted and the power we have always needed.

Frequently used derivations of the ItemsControl class include the ListBox, ListView, and TreeView. The ItemsControl class exposes a wide variety of properties for customizing the look of the control and also of its contained items. Because these properties are exposed as DependencyProperties, they can be data-bound to other properties. These properties include the following:

- **ItemsPanel**—The ItemsControl needs a panel to lay out its children. We specify the panel using an ItemsPanelTemplate. The ItemsPanelTemplate is then applied to an ItemsPresenter.

- **ItemTemplate**—The ItemTemplate is the DataTemplate for the items being displayed. This template may be applied to a ContentPresenter or a ContentControl.

- **ItemContainerStyle**—This property indicates the style for the UI container for each individual item. Note that an ItemControl wraps each data item within a UI container such as a ContentPresenter or a ContentControl-derived class.

- **Template**—This defines the ControlTemplate for the ItemsControl itself.
If this seems like a lot to take in, don’t worry. The concepts behind content controls, presenters, and data templates can seem daunting at first, but we use them so extensively throughout this book that their use will quickly become second nature to you. We cover the `ItemsControl` in greater detail in Chapter 5, “Using Existing Controls,” and Chapter 8, “Virtualization.”

**The UserControl Class**

The `UserControl` class is a container class that acts as a “black box” container for a collection of related controls. If you need a set of three controls to always appear together and be allowed to easily talk to each other, then a likely candidate for making that happen is the `UserControl` class.

Creating your own `UserControl` is an easy first start at creating your own custom controls. It provides the familiar XAML + Code-Behind paradigm that you can use to define your control’s appearance and associated logic. The `UserControl` class derives from `ContentControl` and makes a few additions to `ContentControl`’s stock dependency properties.

The first thing you may notice about a user control is that the control itself cannot receive keyboard focus nor can it act as a Tab stop. This is because in the static constructor for `UserControl`, the `UIElement.Focusable DependencyProperty` and the `KeyboardNavigation.IsTabStop` property have been set to false.

This makes complete sense when you think about the idea that the primary function of a `UserControl` is to wrap a set of related controls and not act as an interactive control on its own.

To make things more clear, let’s take a look at an example. Suppose that you have to create a search bar for your application that looks something like the one in Figure 2.4.

![Search bar](image)

**FIGURE 2.4** A sample interactive search bar for a WPF application.

The search bar in Figure 2.4 is comprised of a `TextBox` and a `Button`. When a user types a keyword or set of keywords and then presses the Enter key, the search functionality is invoked. The same functionality is invoked if the user types in a keyword and clicks the Search button.

While you can place these two controls individually in your window, their purpose and functionality are so interconnected that you would never really use them separately. This makes them ideal candidates for being placed inside a `UserControl`.

To further enhance the encapsulation, you could write your `UserControl` such that it doesn’t tell the hosting container when the user presses Enter or when the user clicks the Search button; it simply exposes a single event called `SearchInvoked`. Your window could
listen for that event and, in an ideal Model-View-Controller world, pass the search request on to a search controller for processing.

Within the UserControl, you have the ability to improve the look and feel of that single element without affecting the UI definition of the window and enabling your control for reuse in multiple locations throughout your application. Additionally, wrapping a set of related controls and giving them a purpose-driven name such as searchBar makes your XAML and your code easier to read, maintain, and debug.

Similar to the way refactoring allows you to incrementally improve your C# code to make it more understandable, maintainable, and testable, refactoring the UI provides the same benefits and is much easier to do within the bounds of a UserControl. This is often called view refactoring.

**The Panel Class**

The Panel class is an element that exists solely to provide the core layout functionality in WPF. Powerful, dynamic layout capability has always been something that was missing in Windows Forms, and now that WPF has dynamic layout features, the world is a much happier place.

Think of the Panel as a “layout brain” rather than something that actually produces its own UI. Its job is to size the child elements and arrange them in the allocated space, but it has no UI of its own. WPF ships with a powerful set of panels that handle many of the common layout scenarios that developers run into on a daily basis. These include the Grid, StackPanel, DockPanel, and the WrapPanel. The following is a brief description of each layout pattern (don’t worry, you see plenty more of these classes in the code samples throughout the book):

- **Grid**—Provides a row/column paradigm for laying out child controls.
- **StackPanel**—Child controls are laid out in horizontal or vertical stacks.
- **DockPanel**—Child controls are docked within the container according to the preferences specified by each child control.
- **WrapPanel**—Child controls in this panel wrap according to the specified wrapping preferences.

Another panel called the Canvas provides static, absolute coordinate-based layout. Panels can be nested within each other to create more complex layouts. Layout in WPF is handled using the two-phased approach of measure and arrange.
During the measure phase, the parent requests that each of its children supply their minimum-required dimensions. The parent then applies additional requirements such as margins, alignment, and padding.

Once each child has been measured, the parent panel then performs the arrange phase. During this phase, the parent panel places each child control in its actual position in the final dimensions. The final position and size of the child element may not be what the child element requested. In these scenarios, the parent panel is the final authority on where the child controls are and how much space they take up.

Panels also have some extra functionality that you might not want to supersede, such as built-in ability to work with ItemsControls and the ability to dynamically change the z-order of a child element with the Panel.SetZIndex method.

The Decorator Class

A Decorator class is responsible for wrapping a UI element to support additional behavior. It has a single Child property of type UIElement, which contains the content to be wrapped. A Decorator can be used to add simple visual decoration, such as a Border, or more complex behavior such as a ViewBox, AdornerDecorator, or the InkPresenter.

When you subclass a Decorator, you can expose some useful DependencyPropertyProperties to customize it. For example, the Border class exposes properties like BorderBrush, BorderThickness, and CornerRadius that all affect how the border is drawn around its child content.

The Adorner Class

If we already have an additive decoration class in the form of the Decorator, why do we need an Adorner class? As mentioned earlier, every single class in the class hierarchy that makes up WPF has a specific purpose. While a Decorator is responsible for drawing decoration around the outside of a piece of child content, the Adorner class allows you to overlay visuals on top of existing visual elements. An easy way to think of adorners is that they are secondary interactive visuals that provide additional means to interact with the primary visual. That might seem complex, but think about widgets such as resizing grips that appear on elements in a typical diagramming program. Those are a secondary visual that sit on top of the elements that they are adorning and provide additional functionality and interaction. By clicking and dragging the resizing-handles, the user can resize the underlying control.

Adorner classes work in conjunction with the AdornerDecorator, which is an invisible surface on which the adorners rest. To be part of the visual tree, adorners have to have a container. The AdornerDecorator acts as this container.

AdornerDecorators are generally defined at the top of the visual tree (such as the ControlTemplate for the Window control). This makes all adorners sit on top of all of the Window content. We explore the use of adorners throughout the book, but you see them specifically in Chapter 6, “The Power of Attached Properties,” and Chapter 9, “Creating Advanced Controls and Visual Effects.”
The Image Class

You might be a little surprised to see the Image class mentioned here among all of the other highly interactive visual controls. In most frameworks, images contain just enough functionality to display rasterized (nonvector) images and maybe support reading and writing streams of image data, but that’s about it.

Image classes can actually provide control-like capabilities for some specific scenarios. Image derives from FrameworkElement, so it can be composed in logical trees and has rich support for event handling and layout. It encapsulates the functionality to render an instance of an ImageSource, specified via the Source property. The ImageSource class can represent a vector image like DrawingImage or a raster/bitmap image like the BitmapSource.

Images can be useful when you want to visualize a large amount of data for which you have limited interaction. Some situations where this might come in handy are when you are visualizing high-volume graphs or network monitoring tools that are visualizing thousands of network nodes. In cases like this, even DrawingVisuals become extremely expensive because each data item is a separate visual and consumes CPU and memory resources. Using an image, and knowing that each data point doesn’t need to be interactive, you can visualize what you need without bringing the host computer to its knees.

Since the Image class also has event handling support, we can attach handlers for mouse events that can query the pixel at the mouse’s current coordinates and report information about that data item. With a little bit of creativity and forethought, the Image class can be a powerful tool in any developer’s toolbox.

The Brushes

The Brush-related classes in WPF represent a powerful way of drawing simple to complex graphics with extreme ease of use. A brush represents static noninteractive graphics that serve mostly as backgrounds on visual elements. You can use a basic brush like SolidColorBrush, which only draws solid colors like Red, Blue, LightGray, and so on, and also gradient brushes like a LinearGradientBrush and RadialGradientBrush. The gradient brushes have additional properties to control the style of drawing the gradient. Figure 2.5 shows you various kinds of gradient brushes.

Although solid and gradient brushes are available in previous UI technologies, the real power comes with the TileBrush classes such as ImageBrush, DrawingBrush, and VisualBrush. An ImageBrush as the name suggests allows you to create a Brush out of an image. This is useful since it allows you to use an image without using the Image class. Since it is a brush, you can use it wherever a Brush type property is expected.

DrawingBrush gives you the power of defining complex graphics as a simple brush. Using DrawingGroups and GeometryDrawings, you can define nested graphics that can provide elegant backgrounds to your visuals. In Figure 2.6, you can see a nested set of graphic elements to create the final DrawingBrush. With clever use of DrawingBrushes, you can simplify the way you define some ControlTemplates.
A `VisualBrush` gives you a live snapshot of a rendered element from the visual tree. We see many uses of `VisualBrushes` in later chapters, such as using `VisualBrush` as a texture on a 3D model or creating reflections.
The TileBrush can also be stretched and tiled to fill the bounds of the visual. You can also cut out a rectangular section of the brush using the Viewport and ViewBox properties. Just like regular visuals, you can also apply transforms. Brushes have two kinds of transform properties: RelativeTransform and Transform. The RelativeTransform property scales the brush using the relative coordinates of the visual ([0,0] to [1,1]). It is useful if you want to transform the brush without knowing the absolute bounds of the visual on which it is applied. The Transform property works after brush output has been mapped to the bounds of the visual—in other words, after the RelativeTransform is applied.

The DataTemplate, ControlTemplate, and ItemsPanelTemplate Classes

WPF has a set of template classes that are used to represent visual trees. Templates are never actually rendered directly; rather, they are applied to other container classes like a ContentPresenter, ItemsPresenter, or a Control.

Each template class derives from the FrameworkTemplate class. These include the DataTemplate, ControlTemplate, and ItemsPanelTemplate classes. There is also a HierarchicalDataTemplate that is used for representing hierarchical data. It takes a little getting used to, but once you are, it is an invaluable tool for representing multilevel or tiered data. HierarchicalDataTemplates are used for controls such as the TreeView.

Each of these three templates contains a visual tree that can be greater than one level. The exception here is that the ItemsPanelTemplate can only contain a Panel-derived class as the root (there is a hint to this exception in the name of the template class itself).

The Viewport3D Class

So far every class that we have discussed so far has been a flat, two-dimensional control. WPF also gives developers unprecedented power and accessibility into the world of 3D programming. The Viewport3D class (see Figure 2.7) gives developers the ability to work in three dimensions without having to deal with complex game-oriented frameworks such as Direct3D or OpenGL.

The Viewport3D class is a container for a 3D world that is comprised of 3D models, textures, cameras, and lights. Viewport3D derives from the FrameworkElement class instead of Control. This makes a good deal of sense because FrameworkElement works great as a visual container, and the Viewport3D class is a visual container for an interactive 3D scene.

The Viewport3D class also has no background. As a result, you can place a 3D viewport on top of 2D elements and create stunning effects by mixing and matching 2D and 3D visual elements. Just keep in mind that the 3D world must reside in a completely different container. For example, you can use a VisualBrush to take a 2D visual and apply it to the surface of a 3D model as a material. The .NET Framework 3.5 introduced additional classes that allow you to have live, interactive 2D visuals on a 3D surface. For example, you can place a Button visual as a material for a Sphere and interact with it like a regular button, even if the Sphere is spinning and being dynamically lit by a light source.
The **MediaElement** Class

Many of today’s modern applications are more than just static controls and grids and buttons. Many of them contain multimedia such as sounds, music, and video. WPF not only lets you play audio and video, but gives you programmatic control of the playback.

WPF gives you this multimedia programming experience with the MediaElement class. You indicate the source of the media using the Source property. You can control the media playback using the Play, Pause, and Stop methods. You can even control the volume and skip to a specific time in the playback using the Position property.

Figure 2.8 shows a simple WPF application that contains a media element and some controls for manipulating the video.

The **InkCanvas**

The Tablet PC introduced a more widespread use of the stylus as a means to interact with the applications. The strokes created using the stylus were treated as ink, which could also be mapped to application-specific gestures. Although the stylus is treated as the default device, a mouse makes a good substitute.
WPF has the **InkCanvas** class that provides most of the features available on the Tablet PC. In fact the **InkCanvas** becomes the slate on which we can scribble either with the mouse or with the stylus. **InkCanvas** can be created declaratively and exposes a variety of events related to strokes. It also has built-in gesture recognition for some standard gestures. By overlaying an **InkCanvas** on some UI elements, you can add some interesting features to an application. For example, for a photo-viewing application, you can overlay an **InkCanvas** on a photo to annotate parts of the picture, as shown in Figure 2.9.

![Figure 2.9](image)

**FIGURE 2.9** A simple **InkCanvas** application.

**Summary**

With the diverse range of classes available to use within WPF, we can see that WPF is a great toolset for creating interesting, compelling, and visually stunning interfaces and controls.

Understanding and respecting this diversity and knowing which is the best tool for any given situation will make your development experience more rewarding and enjoyable and will ultimately improve the quality of your applications.

Table 2.1 presents a summary of the classes discussed in this chapter.

In the next chapter, we discuss creating controls and some best practices for approaching control creation. We build on the foundations from this chapter and the previous chapter as we venture into the world of creating exciting, modern interfaces with WPF.
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DispatcherObject</td>
<td>Provides threading and messaging control</td>
</tr>
<tr>
<td>DependencyObject</td>
<td>Provides hosting for dependency properties, a core building block for many of WPF’s advanced features, such as data binding</td>
</tr>
<tr>
<td>Visual, DrawingVisual</td>
<td>Provide functionality for drawing simple graphics</td>
</tr>
<tr>
<td>Shape</td>
<td>Provides functionality for drawing geometry-based graphics</td>
</tr>
<tr>
<td>Decorator</td>
<td>Draws decorations around a contained UIElement</td>
</tr>
<tr>
<td>Adorner</td>
<td>Provides a decorative overlay on top of other controls</td>
</tr>
<tr>
<td>FrameworkElement</td>
<td>A container for other visual elements</td>
</tr>
<tr>
<td>GlyphRuns, TextBlock, FlowDocument</td>
<td>Provide text support with increasingly complex set of features for formatting and layout</td>
</tr>
<tr>
<td>Panel</td>
<td>The “layout brain” container for other elements</td>
</tr>
<tr>
<td>Control</td>
<td>A UI component that supports templating</td>
</tr>
<tr>
<td>ContentPresenter</td>
<td>Container for holding a DataTemplate</td>
</tr>
<tr>
<td>ItemsPresenter</td>
<td>Container for holding an ItemTemplate</td>
</tr>
<tr>
<td>ContentControl</td>
<td>A control that contains a single child</td>
</tr>
<tr>
<td>ItemsControl</td>
<td>A control that displays a list of items</td>
</tr>
<tr>
<td>UserControl</td>
<td>A “black box” container for multiple logically and visually related controls</td>
</tr>
<tr>
<td>DataTemplate, ControlTemplate, ItemsPanelTemplate</td>
<td>Reusable visual tree templates</td>
</tr>
<tr>
<td>Image</td>
<td>A high-performance graphics control for high-volume data visualization or display of raster and vector images</td>
</tr>
<tr>
<td>Brush</td>
<td>Provides static graphics for backgrounds, ranging from the simple SolidColorBrush to the complex TileBrushes</td>
</tr>
<tr>
<td>Viewport3D</td>
<td>A container for an interactive 3D world</td>
</tr>
<tr>
<td>MediaElement</td>
<td>Plays audio and/or video within a container</td>
</tr>
<tr>
<td>InkCanvas</td>
<td>For creating Tablet PC-like strokes and gestures</td>
</tr>
</tbody>
</table>
Numbers
2D bounds, 3D objects, 198-199
2D visuals, 3D surfaces, mapping on, 192-199
2D-on-3D surfaces, interactivity, 200-201
3D layout, 200
3D mesh geometries, animations, 214
3D objects, 2D bounds, 198-199
3D Programming for Windows, 215
3D surfaces
  2D visuals, mapping on, 192-199
  creating, 219-220
3D virtualization, 140-142
3D worlds, 185-186
  cameras, 186-187
  ContainerUIElement, 196-199
  lights, 186-187
  models, 186-187
  ModelUIElement, 196-199
  performance, 325-326
  rendered output, XAML, 188
  Viewport3D element, 186-192

A
Abrams, Brad, 304
  absolute layout, 85
    abstract class, 13
actions. See data
AddOwner() method, DPs (dependency properties), 279-280
Adobe Illustrator, 309, 312
Adorner class, 24, 29
Aero theme, 168
affine transformations, 66
angle of rotation, 42
animated scrolling, 122-123
animations, 203, 221
  3D surfaces, creating, 219-220
  CompositionTarget.Rendering, 204-206
  custom animations, creating, 212-220
  dependency properties, 206
  DispatcherTimer, 204
  DrawingContext, 220-221
  interpolation, 216
    Bézier interpolation, 214
      linear interpolation, 214, 218
keyframe animations, 207-209
layout animations, attached properties, 102
path animations, 213
path-based animations, 211-212
pixel shaders, 235
procedural animations, 203-206
repeating animations, 329
storyboards, 206-212, 323
type-based animations, 206-207
APIs (application programming interfaces)
automation APIs, 338-342
TestAPI library, 349
application services, attached properties, 102
ApplicationIdle priority level (Dispatcher), 286
applications. See performance; UI automation
applying transitions, 159-161
ApplyTemplate override, runtime skinning, 177-182
ApplyTransition method, 159-161
APs (attached properties). See attached properties
arc shapes, 41
creating, 42-44
Arrange call, 51-52
assemblies, UI automation, 333
attached events, routed events, 246-248
attached properties, 35-38, 93-95, 111-112
application services, 102
as extension points, 100-103
constraining panels, 102
controls, 297
custom panels, creating, 58-63
data templates, 102
dependency properties, compared, 93
DnD (drag and drop), implementing, 103-111
ink-bases functionality, encapsulating as, 146
layout animations, 102
persistent ToolTips, creating, 97-98
property abstraction, 102
Timeline.DesiredFramerate, 220
UI helper objects, 103
UseHover attached property, building, 95-100
AttachToVisualTree method, 182
automation. See also UI automation
custom controls, 343-349
automation APIs, 338-342
automation elements, performing operation
on, 342
automation tree
locating elements in, 339-340
navigating, 342
control patterns, checking for, 340
events, listening to, 341-342
property values, reading, 341
automation tree
elements, locating in, 339-340
navigating, 336-338, 342
AutomationElement class, UI automation, 333-335
AutomationPeer class, UI automation, 333-335

B
Background priority level (Dispatcher), 286
background threads, long-running operations, 322
BackgroundWorker class, data binding, 289-291
base peer classes, selecting, 343-344
Beck, Kent, 303
Behavior class, 101-102
behaviors, 3, 6-8, 34
Bézier interpolation, 214
Binding class, 284-285
blocking calls, 107
Blois, Pete, 312
Border properties, 71
bounds
2D bounds, 3D objects, 198-199
scrolling regions, controlling, 116
Brush class, 29
Brush-related classes, 25-27
brushes, 320
customizing, 82-83
gradient brushes, 306-307
tile brushes, 304-305
XAML coding conventions, 307-308
builder design pattern (GUI development), 301-302
Button controls, customizing, 32

caching, performance, 320
calls
Arrange, 51-52
blocking calls, 107
Measure, 51-52
synchronous calls, 107
cameras
3D scenes, 186
3D worlds, 186-187
CanExecute method, 255
Canvas class, 23
Canvas panel, 319
child elements, controls, strong references, 297
Child property (Decorator class), 24
children
layout containers, conversations, 51-52
visual children, custom panels, 52-55
Children property, Panel class, 53
circular minute timers, building, 40-46
class handlers
registering, 249
routed events, 249-250
classes, 5, 11. See also visual classes
  base peer classes, selecting, 343-344
  geometry classes, 18
  partial classes, defining controls as, 296
  UI virtualization, 131-132
CLINQ (Continuous LINQ), 291-292
clip geometries, 310-312
cloning resources, xShared, 299-300
coection, properties, 277
collinearity ratios, 66
CombinedGeometry class, 18
CommandBinding, 257
commands, 7, 255-258
  components, 257
  events, compared, 259-261
  ICommandSource interface, 262-266
  request requery, 261-262
  routed commands, 259
CommandSource, 257
CommandTarget, 258
component interaction, UI virtualization, 132-133
ComponentResourceKey, 172
composed method design pattern (GUI development), 303
composite data, 2
composite data-based controls, 34
CompositionTarget.Rendering, animations with, 204-206
concentric circles, creating, 85-91
constraining panels, attached properties, 102
container styles, customizing, 74-75, 77
containers
  container recycling, 323
  IoC (Inversion of Control) containers, 298
  layout containers, conversations, 51-52
  navigation containers, 271
  recycling, 140, 323
  SectorVisual class, creating for, 15
  UI virtualization, 130-131
ContainerUIElement3D, 196-199
ContainerVisual class, 13
ContentControl class, 20, 29, 156
ContentPresenter class, 20-21, 29
ContextIdle priority level (Dispatcher), 286
Continuous LINQ (CLINQ), 291-292
Control class, 19-20, 29
control patterns
  checking for, 340
  selecting, 344-345
  UI automation, 333-335
control templates, 4
controls, 69-70, 166. See also names of specific controls
  attached properties, 297
  child elements, strong references, 297
  composite data-based controls, 34
  creating, 143
  DockSlidePresenter, 146-154
  lasso selection tool, 143-146
  TransitionContainer, 154-161
  custom controls
    automation of, 343-349
    creating, 40-47
    layered structures, 39
    needs assessment, 31-39
  customizing, 32, 70-82
    behavior extensions, 34
    brushes, 82-83
    ControlTemplates, 70-72
    data transformations, 32-34
    DataTemplates, 71-72
    ListBox, customizing, 74-78
    properties, 70
    property exposure, 182-183
data shapes, 34
designing, 295-301
docking, 149-154
hierarchy-based controls, 34
internal resource dictionaries, 295-296
internal state management, scoped DPs, 296
ItemsControl, customizing, 72-74
list-based controls, 34
ListBox, customizing, 74-78
lookless controls, 32, 70
low-priority delegates, 299
markup extensions, 300-301
multiple events, handling, 299
partial classes, defining as, 296
performance. See performance properties, attached properties, 35-38
property changes, handling, 299
range-based controls, 34
resources, cloning, 299-300
singular controls, 34
subparts, communicating between, 297-298
  templates, primitive visual identification, 297
UI (user interface), visual tree, 332
UI automation. See UI automation undocking, 149-150, 152-154
virtualized controls, creating, 135-139
ControlTemplate, 81, 91, 177
ControlTemplate class, 27, 37-38
  controls, customizing, 32, 70-72
  ProgressBar control, extending, 43-46
ControlTemplate for the Circular Minute Timer listing (3.4), 44-46
ControlTemplate for the process workflow Progressbar listing (3.2), 37-38
conversations, layout, 51-52
converters, 5
  TypeConverters, data transformations, 33-34
  value converters, data transformations, 32-33
Creating a Container for the SectorVisual Class listing (2.2), 16
Creating the Arc Geometry listing (3.3), 42-43
custom animations, creating, 212-220
custom controls
  automation of, 343-349
  creating, 40-47
  layered structures, 39
  needs assessments, 31-39
custom panels
  creating, 49-55
    attached properties, 58-63
    custom scrolling, 117-122
    transformations, 63-68
    VanishingPointPanel, 56-58
    WeightedPanel, 58-63
  layout, 49-52
  visual children, 52-55
custom pixel shaders
  animating, 235
  grayscale pixel shaders, writing, 228-231
  GUI interaction, 235-239
  multi-input shaders, 239-241
  parameterized pixel shaders, writing, 231-235
  Shazzam tool, 242
  writing, 228-242
custom ScrollBar, creating, 78-82
custom scrolling, custom panels, building, 117-122
customizing
  controls, 32, 70
    behavior extensions, 34
    brushes, 82-83
    ControlTemplates, 70-72
    data transformations, 32-34
    DataTemplates, 71-72
    ItemsControl, 72-74
    ListBox, 74-78
    ListBoxes, 83-91
    properties, 70
    ScrollBar, 78-82
    ItemContainerStyle, 74-77
    ItemsPanelTemplate, 77-78
    ItemTemplate, 77-78
Cwalina, Krzysztof, 304

D
data, 2-3
  composite data, 2
  hierarchical data, 2
  list data, 2
  primitive data, 2
data binding, 4, 275
  Dispatcher class, 285, 287
    BackgroundWorker class, 289-291
    deferring UI operations, 287
    posting messages from worker threads, 287-289
  DPs (dependency properties), 276
    AddOwner method, 279-280
    listening to property changes on, 280-282
    precedence, 276-279
    evolution of, 275
    NotifyOnSourceUpdated property, 284-285
    NotifyOnTargetUpdated property, 284-285
    performance, 321-322
    RelativeSource.PreviousData property, 282-284
data flow, 6
data shapes, controls, 34
data templates, 4, 102
data transformations
  controls, customizing, 32-34
  TypeConverters, 33-34
  value converters, 32-33
data types, 2-3
  HLSL (High Level Shading Language), 226-227
DataBind priority level (Dispatcher), 286
DataTemplate class, 27-29, 91, 183, 319
  controls, customizing, 71-72
Decorator class, 24, 29
Default (theme) Styles precedence level (DPs), 277-278
default display form, ProgressBar control, 35
default styles
  building, 169-170
  resources in, 170-172
defered scrolling, 139-140, 329
DefiningGeometry property (Shape class), 16-18
dependency properties. See DPs (dependency properties)
DependencyObject class, 12, 16, 29
design patterns
  GUI development, 301, 304
    builder pattern, 301-302
    composed method pattern, 303
    factory method pattern, 303
    MVC (model-view-controller) pattern, 302
MVVM (model-view-view-model)
pattern, 302
state pattern, 303
strategy pattern, 301
weak events, 250-255

Design Patterns in C#, 301
designing. See also visual design
controls, 295-301
philosophy, data and behaviors, 1-3
tools, 312-314

Dingbats font, icons, using for, 308-309
DirectX, pixel shaders, 223
Dirty Rect Addition Rate category
(Perforator), 328
Dispatcher class, data binding, 285-287
  BackgroundWorker class, 289-291
deferring UI operations, 287
  posting messages from worker threads, 287-289
DispatcherObject class, 12, 16, 29
DispatcherTimer, 204
DisplacementFilter, 239-241
display form (ProgressBar control), 35
DnD (drag and drop)
  drag sources, 104-105
  drop targets, 104-105
  events, 103
  implementing attached properties, 103-111
DockFrontChild() method, 150-152
Docking controls, 149-154
DockPanel, 23, 59-60, 71, 319
DockSidePresenter control, creating, 146-154
DoDragDrop method, 103
DPs (dependency properties)
  animations, 206
  attached properties, compared, 93
  data binding, 276
    AddOwner method, 279-280
    listening to property changes on, 280-282
    precedence, 276-279
  scoped DPs, internal state management, 296
drag and drop. See DnD (drag and drop)
drag sources (DnD), 104-105
DragDrop.DoDragDrop method, 104
DrawingBrushes, 25-26, 82
DrawingContext, 14-15, 220-221
DrawingGroups class, 25
DrawingVisual class, 13-15, 29
DrawingVisual control, 70
drop shadows, implementing, 163
drop targets (DnD), 104-105
Dussud, Cedric, 127
dynamic skinning, 174-177

E
effect mapping, pixel shaders, GUI interaction, 235-239
effects, 166
drop shadows, implementing, 163
gloss effects, implementing, 164-165
opacity masks, implementing, 164
reflection effect, implementing, 161-162
transitions
  applying, 159-161
  creating, 154-157
  handling, 157-159
embedded fonts, icons, 308-309
embedding, ViewPort3D element, 189-192
encapsulation, object creation, markup
extensions, 300-301
enumeration values, TreeScope, 337
environments
  3D environments, 185-192
  pixel shaders, setting up, 224-227
event triggers, routed events, 245-246
events, 255. See also routed events;
  weak events
    commands, compared, 259-261
    drag and drop, 103
    focus, 267-270
    layout, 66-68
    listening to, 341-342
    multiple events, handling, 299
    property changed events, listening to, 280-282
Expression Blend, 312
extending
  behaviors, controls, 34
  ProgressBar control, 40-46
extension points, attached properties as, 100-103
ExtractElement method, 111

F
faces, 3D shapes, 186
factory method design pattern (GUI
development), 303
Ferris Wheel rotations, 63
FinishTransition() method, 161
flip scrolls, 122
flow control, HLSL (High Level Shading
Language), 227
FlowDocument class, 18-19, 29
FluidKit project, 161, 200

How can we make this index more useful? Email us at indexes@samspublishing.com
focus, 266
  events, 267-270
  keyboard focus, 266
  keyboard navigation, 271-273
  logical focus, 266-267
  properties, 267-270
  scopes, 266-267
  visual styles, 268
Focus() method, 266
FocusVisualStyle, 268
fonts, nonstandard fonts, icons, 308-309
Frame Rate category (Perforator), 328
frame rates, 186
Framework Design Guidelines, 304
framework property metadata options, 323-324
FrameworkElement class, 15-16, 29, 156
FrameworkTemplate class, 27
freezables, 321-322
frustums, 186

G

Geometries, 16, 310-312
gometry classes, 18
GeometryDrawings class, 25
GeometryGroup class, 18
GetChild() method, 83
GetChildren() method, 83
GetChildrenCount() method, 83
GetContentBounds() method, 83
GetEffect() method, 83
GetOpacity() method, 83
GetParent() method, 83
GetTemplateChild() method, 182
GetTransform() method, 83
GetValue() method, 276
GetVisualChild method, 53
GetVisualFeedback() method, 110
glyph effects, implementing, 164-165
GlyphRuns class, 18-19, 29
Goldberg, Jossef, 321-322, 326
GPU (Graphics Processing Units), shaders, 223
gradient brushes, 306-307
gayscale pixel shaders, writing, 228-231
Grid panel, 23, 59-60, 71, 319
GUI development
design patterns, 301
  builder pattern, 301-302
  composed method pattern, 303
  factory method pattern, 303
  framework design, 304
  MVC (model-view-controller) pattern, 302
  MVVM (model-view-view-model) pattern, 302
  state pattern, 303
  strategy pattern, 301
  pixel shaders, effect mapping, 235-239

H

handling transitions, 157-159
hardware rendering, 324
helper objects, UI controls, attached
  properties, 103
hierarchical data, 2
hierarchy, visual classes, 11-12
hierarchy-based controls, 34
HitTest() method, 83, 145
HLSL (High Level Shading Language), 223-225
data types, 226-227
  flow control, 227
  intrinsic functions, 227
  keywords, 226
  pixel shaders
    environment setup, 224-227
    writing, 228-242
  samplers, 227
  semantics, 227
textures, 227
Hoffman, Kevin, 292
HookUpCommand() method, 264
horizontal scrolling, user-requested scrolling,
  responding to, 116
HoverInteractor class, UseHover attached
  property, building, 95-100

I

ICommand, 255-258
  request requery, 261-262
  routed commands, 259
ICommandSource interface, 262-266
icons, nonstandard fonts, 308-309
IDropTargetAdvisor method, 110
Illustrator, 309, 312
Image class, 25, 29
ImageBrush class, 25
Implementation Patterns, 303
Implicit Style precedence level (DPs), 277-278
importing PSD files, 309
Inactive priority level (Dispatcher), 286
Ingebretsen, Robby, 313
Inheritance precedence level (DPs), 277-278
ink-based functionality, attached properties, encapsulating as, 146
InkCanvas class, 28-29, 143-146
INotifyPropertyChanged, 282
input devices, evolution of, 7
Input priority level (Dispatcher), 286
InputBinding class, 266
interactive search bars, 22
interactivity, 2D-on-3D surfaces, 200-201
interfaces, 2
GUI development
  design patterns, 301-304
  pixel shaders, effect mapping, 235-239
  UIs (user interfaces), 2-3
  commands, 255-266
  deferring operations, Dispatcher, 287
  routed events, 243-255
  visual tree, 332
internal resource dictionaries, 295-296
internal state management, scoped
  dependency properties, 296
interpolation, 216
  Bézier interpolation, 214
  linear interpolation, 214, 218
intrinsic functions, HLSL (High Level Shading Language), 227
Invalid priority level (Dispatcher), 286
Inversion of Control (IoC) containers, 298
IoC (Inversion of Control) containers, 298
IScrollInfo, 115-116, 123-124, 127, 322-323
  bounds, controlling, 116
  custom scrolling functionality, adding, 119-122
  Thumb
    location management, 116
    logical scrolling, 117
    user-requested scrolling, responding to, 116
  IsValidDataObject method, 110
ItemContainerGenerator, UI virtualization, 132-135
ItemContainerStyle, customizing, 74-77
ItemContainerStyle property (ItemsControl class), 21
ItemsControl class, 21-22, 29
  customizing, 72-74
  ItemContainerStyle, 21
  ItemsPanel, 21
  ItemTemplate, 21
  RelativeSource.PreviousData property, 284
  Template, 21
  UI virtualization, 131-133
ItemsControl control, 70
ItemsPanel property (ItemsControl class), 21, 131-133
ItemsPanelTemplate class, 27-29, 77-78
ItemsPresenter class, 29
ItemTemplate, customizing, 77-78
ItemTemplate property (ItemsControl class), 21
IWeakEventListener, weak events, delivering, 254-255

J–K
JetBrains DotTrace, 329
Kaxaml tool, 313-314
keyboard focus, 266
  events, 267-270
  properties, 267-270
keyboard navigation (focus), 271-273
Keyboard.Focus() method, 266
keyframe animations, 207-209
keywords, HLSL (High Level Shading Language), 226
Kutruff, Andrew, 292

L
Language Integrated Query (LINQ), 291
lasso selection tools, creating, 143-146
layered structures, custom controls, 39
layout, 5, 49-52
  3D layout, 200
  absolute layout, 85
  circular layouts, 63
  conversations, 51-52
  events, 66, 68
  layout logic, custom scrolling, 117-119
  layout space, 51
  transformations, 63-68
  UIElement class, 5, 50
layout animations, attached properties, 102
layout logic, custom scrolling, creating, 117-119
layout patterns, 23
layout space, 51
LayoutTransforms, 66-67
light
  3D worlds, 186-187
  refraction, 187
light sources, 3D scenes, 186
line charts, RelativeSource.PreviousData property, 283
linear interpolation, 214, 218
LinearGradientBrush class, 25-26
LINQ (Language Integrated Query), 291
list data, 2
list-based controls, 34

How can we make this index more useful? Email us at indexes@samspublishing.com
ListBoxes, customizing, 74-78, 83-91

listings

2.1 (Pizza Slice” Sector Visual Class), 13
2.2 (Creating a Container for the SectorVisual Class), 16
2.3 (Making the SectorVisual into a Shape), 17
3.1 (ProcessStageHelper Class), 36
3.2 (ControlTemplate for the process workflow ProgressBar), 37-38
3.3 (Creating the Arc Geometry), 42-43
3.4 (ControlTemplate for the Circular Minute Timer), 44-46

Loaded priority level (Dispatcher), 286
Local Values precedence level (DPs), 277-278

logical focus, 266

- events, 267-270
- focus scopes, 266-267
- properties, 267-270

logical scrolling, 117

LogicalTreeHelper class, controls, customizing, 82-83

long-running operations, background threads, 322

lookless controls, 32, 70

low-priority delegates, controls, 299

M

Magnifying Glass control, 154

Making the SectorVisual into a Shape listing (2.3), 17

mapping 2D visuals, 3D surfaces, 192-199

Mariani, Rico, 326

markup extensions, object creation, encapsulating, 300-301

Measure call, 51-52

MeasureOverride method, 135

measuring performance, 326-327

- Perforator, 328-329
- third-party tools, 329
- Visual Profiler, 327-328

MediaElement class, 28-29

MergedDictionaries, 171

mesh surfaces, vertices, 219

mesh vertices, 148

messages, worker threads, posting from, 287-289

methods

- AddOwner(), 279-280
- ApplyTransition(), 159, 161
- AttachToVisualTree(), 182
- CanExecute(), 255
- DockFrontChild(), 150, 152
- DoDragDrop(), 103
- DragDrop.DoDragDrop(), 104
- ExtractElement(), 111
- FinishTransition(), 161
- Focus(), 266
- GetChild(), 83
- GetChildren(), 83
- GetChildrenCount(), 83
- GetContentBounds(), 83
- GetEffect(), 83
- GetOpacity(), 83
- GetParent(), 83
- GetTemplateChild(), 182
- GetTransform(), 83
- GetValue(), 276
- GetVisualChild(), 53
- GetVisualFeedback(), 110
- HitTest(), 83, 145
- HookUpCommand(), 264
- IDropTargetAdvisor(), 110
- IsValidDataObject(), 110
- Keyboard.Focus(), 266
- MeasureOverride(), 135
- OnDropCompleted(), 106, 110
- OnRender(), 86
- PrepareStoryboard(), 159, 161
- PrepareStoryboard(), 152-153, 158
- Scroll(), 122
- SetupVisuals(), 159, 161
- SetupVisuals(), 157-158
- SetValue(), 276
- SetVerticalOffset(), 119, 121
- Stroke.GetBounds(), 144
- UndockFrontChild() method, 150, 152
- UpdateScrollInfo(), 121

Metsker, Steven John, 301

minute times (circular), building, 40-46

models, 3D worlds, 186-187

ModelUIElement3D, 196, 198-199

Mole tool, 313-314

Morrison, Vance, 326

multi-input pixel shaders, 239-241

multiple events, handling, state machines, 299

MVC (model-view-controller) design pattern, 309

MVVM (model-view-view-model) design pattern, 302

N

namespaces

- System.Windows namespace,
  DependencyObject class, 13
- System.Windows.Threading,
  DispatcherObject class, 12
- UI automation, 333
navigation containers, 271
needle-shaped DrawingBrushes, 82
nonstandard fonts, icons, 308-309
Normal priority level (Dispatcher), 286
NotifyOnSourceUpdated property, data binding, 284-285
NotifyOnTargetUpdated property, data binding, 284-285

O
objects
encapsulation, markup extensions, 300-301
provider objects, 298
target objects, 35
OnDropCompleted method, 106, 110
OnRender method, 86
opacity masks, 310
clip geometries, compared, 310
implementing, 164
Open Source FluidKit project, 200
OriginalSource, Routed EventsArgs, 246

P
painter's algorithms, 319
Panel class, 23-24, 29
Children property, 53
layout patterns, 23
VisualChildrenCount property, 53
panels, 5, 70. See also names of specific panels
Canvas, 319
constraining panels, attached properties, 102
custom panels
creating, 49-68
custom scrolling, 117-122
layout, 49-52
visual children, 52-55
DockPanel, 319
Grid, 319
performance, 319
virtualized panels, creating, 135-139
Parallax view control, 154
parallel timelines, storyboards, 208-209
parameterized pixel shaders, writing, 231-235
partial classes, controls, defining as, 296
passes, layout conversations, 51
path animations, 213
path-based animations, 211-212
PathGeometry class, 18
Perforator, 328-329
performance, 317-318, 330
3D worlds, 325-326
background threads, long-running operations, 322
brushes, 320
caching, 320
data binding, 321-322
framework property metadata options, 323-324
freezables, 321-322
hardware rendering, 324
measuring, 326-329
perceived responsiveness, 329-330
performance enhancements, 318
pixel shaders, 323
reference handling, 321
render pipeline, optimizing, 325
resource management, 321
scrolling, 322-323
software rendering, 324
storyboard animations, 323
virtualization, 322-323
visuals, 318-319
persistent ToolTips, creating, attached properties, 97-98
perspective cameras, 148
perspective projections, 141
Petzold, Charles, 214
Photoshop, 309, 312
pie shapes, 41
pixel shaders, 223-224, 242
animating, 235
custom pixel shaders, writing, 228-242
DirectX, 223
environments, setting up, 224-227
grayscale pixel shaders, writing, 228-231
GUI interaction, effect mapping, 235-239
HLSL, 225
data types, 226-227
flow control, 227
intrinsic functions, 227
keywords, 226
samplers, 227
semantics, 227
HLSL (High Level Shading Language), 223
improvements, 224
multi-input pixel shaders, 239-241
parameterized pixel shaders, writing, 231-235
performance, 323
Shazzam tool, 242
vertex shaders, compared, 223
“Pizza Slice” Sector Visual Class listing (2.1), 13
PNGs (portable network graphics), transparent PNGs, 309

How can we make this index more useful? Email us at indexes@samspublishing.com
posting messages from worker threads, 287-289
precedence, DPs (dependency properties), 276-279
PrepareStoryboard() method, 152-153, 158-161
presenters, 4
primitive data, 2
primitive visuals, templates, identifying, 297
priorities, Dispatcher class, 285-287
procedural animations, 203-206
ProcessStageHelper class, 36-37
ProcessStageHelper Class listing (3.1), 36
progress bars, 35, 329
ProgressBar control, 35, 40
  default display form, 35
  extending, 40-46
properties
  attached properties, 93-95, 111-112
    application services, 102
    as extension points, 100-103
    constraining panels, 102
    controls, 297
    custom panel creation, 58-63
    data templates, 102
    DnD (drag and drop) implementation, 103-111
    layout animations, 102
    property abstraction, 102
    UI helper objects, 103
    UseHover attached property, 95-100
  automation properties, 335-336
  coercion, 277
  controls
    attached properties, 35-38
    customizing, 70
    customizing with, 32
  dependency properties, 93
    animations, 206
    data binding, 276-282
    internal state management, 296
  focus, 267-270
  framework property metadata options, 323-324
  property mirrors, routed events, 246
  RelativeSource.PreviousData, data binding, 282-284
property abstraction, attached properties, 102
property changes
  DPs (dependency properties), listening to, 280-282
  handling, state machines, 299
property exposure, controls, customizing, 182-183
property mirrors, routed events, 246
Property Value Coercion precedence level (DPs), 277-278
property values
  reading, 341
  TemplateBinding, ControlTemplate, 81
PropertyDescriptor, 281
provider objects, 298
PSD files, importing, 309

Q–R
radar screens
  concentric circles, creating, 85-91
  creating, 83-91
  moving enemies, 84-85
  sweeping cones, creating, 85-91
RadialGradientBrush class, 25-26
range-based controls, 34
RangeSelector custom control, automation, 343-349
recycling (container), 323
Red Gate Ants Profiler, 329
reference figures, 87
reference handling, 321
references, control child elements, maintaining, 297
reflection effect, implementing, 161-162
refraction, light, 187
registering class handlers, 249
relative transforms, 306-307
RelativeSource.PreviousData property
  data binding, 282-284
  ItemsControl, 284
render pipeline, optimizing, 325
Render priority level (Dispatcher), 286
RenderCapability, 324
rendering
  hardware rendering, 324
  render pipeline, optimizing, 325
  software rendering, 324
RenderTransforms, 65-66
repeating animations, 329
resource dictionaries, 295-296
resource keys, dynamic skinning, 176
resource lookups, 168-169
resource management, 321
ResourceDictionaries, 171, 173
ResourceDictionary, 168
resources
  cloning, xShared, 299-300
  default styles, 170-172
retained-mode graphics systems, 15
Ritscher, Walt, 242
RotateTransforms, 66
routed events, 243-245
attached events, 246-248
class handlers, 249-250
event triggers, 245-246
property mirrors, 246
weak events, 250-255
RoutedEventsArgs, 246
RowsPanels, creating, 117-122
runtime skinning, enabling, 174-182

samplers, HLSL (High Level Shading Language), 227
Schechter, Greg, 255
SciTech .NET Memory Profiler, 329
scoped dependency properties, internal state managements, 296
Scroll method, 122
ScrollBars, 40, 113-115
  custom ScrollBar, creating, 78-82
  vertical ScrollBars, 79
scrolling, 113, 127
  animated scrolling, 122-123
  bounds, controlling, 116
  custom scrolling, 117-122
  deferred scrolling, 139-140, 329
  flip scrolls, 122
  IScrollInfo, 115-117, 123-124
  logical scrolling, 117
  performance, 322-323
  ScrollViewer, 124-126
  Thumb, location management, 116
  user-requested scrolling, responding to, 116
scrolling regions, bounds, controlling, 116
ScrollViewer, 117, 124-127, 131-133
SectorVisual class, containers, creating for, 15
semantics, HLSL (High Level Shading Language), 227
Send priority level (Dispatcher), 286
SetupVisuals() method, 157-158, 161
SetValue() method, 276
SetVerticalOffset() method, 119, 121
shaders, 223. See also pixel shaders
  GPUs (Graphics Processing Units), 223
  HLSL (High Level Shading Language), 223-225
  data types, 226-227
  flow control, 227
  intrinsic functions, 227
  keywords, 226
  samplers, 227
  semantics, 227
  vertex shaders, 223
Shape class, 16-18, 29
  DefiningGeometry property, 16-18
  Stretch property, 17
shapes, 63, 70
  3D shapes, faces, 186
  arc shapes, 41-44
  concentric circles, creating, 85-91
  pie shapes, 41
  properties, 71
  traditional circles, 63
Shazzam tool, pixel shaders, 242
Shifflett, Karl, 313
skins, 167-170, 183
  default styles
    building, 169-170
    resources in, 170-172
    resource lookups, 168-169
  runtime skinning
    ApplyTemplate override, 177-182
    enabling, 174-182
  theme-specific styles, creating, 172-174
  themes, compared, 168
SkinThemeControl, 169
Smith, Andrew, 313
Smith, Josh, 313
Snoop tool, 312-313
software rendering, 324
SolidColorBrush class, 25, 320
Source class, 35, 246
Spine, 16
StackPanel class, 23
  layout, 50
  properties, 71
“staged” progress bars, custom display, 35
StaggeredPanel virtualized control, creating, 135-139
state design pattern (GUI development), 303
state machines, 299
state management, scoped dependency properties, 296
storyboard animations, performance, 323
Storyboard Animations precedence level (DPs), 277-278
storyboards
  animating with, 206-212
  docked/undocked modes, 152-153
strategy design pattern (GUI development), 301
StreamGeometry class, 18
Stretch property (Shape class), 17
Stroke.GetBounds method, 144
Style Setters precedence level (DPs), 277-278
Style Triggers precedence level (DPs), 277-278

How can we make this index more useful? Email us at indexes@samspublishing.com
styles, 5-6
  default styles
    building, 169-170
    resources in, 170-172
  focus, visual styles, 268
  resource lookups, 168-169
  runtime skinning, enabling, 174-177
  theme-specific styles, creating, 172-174
subclasing, WEM (WeakEventManager), 252-254
subparts, controls, communicating between, 297-298
surfaces
  2D-on-3D surfaces, interactivity, 200-201
  3D surfaces
    creating, 219-220
    mapping 2D visuals to, 192-196
  mesh surfaces, vertices, 219
SW/HW IRTs per Frame category
  (Perforator), 329
sweeping cones, creating, 85-91
synchronous calls, 107
System.Windows namespace,
  DependencyObject class, 13
System.Windows.Threading namespace,
  DispatcherObject class, 12
SystemIdle priority level (Dispatcher), 286

T

target objects, 35
Template property (Control class), 19
Template property (ItemsControl class), 21
Template Triggers precedence level (DPs), 277-278
TemplateBinding, property values,
  ControlTemplate, 81
TemplateParent Template Properties
  precedence level (DPs), 277-278
templates, 3
  control templates, 4, 70-72
  data templates, 4, 71-72
  primitive visuals, identifying, 297
TestAPI library, 349
Text, performance, 319
text boxes, watermarked text boxes, 269
Text classes, 18-19
TextBlock class, 18-19, 29, 71, 319
TextBlock control, 70
textures, HLSL (High Level Shading Language), 227
theme-specific styles, creating, 172-174

themes, 167-170, 183
  Aero theme, 168
  default styles
    building, 169-170
    resources in, 170-172
  resource lookups, 168-169
  skins, compared, 168
  theme-specific styles, creating, 172-174
Themes folder, 173
third-party tools, performance measuring, 329
Thumb, bounds, controlling for, 116
tile brushes, 304-305
TileBrush classes, 25, 27
Timeline.DesiredFrameRate attached property, 220
tools, design tools, 312-314
ToolTip, persistent ToolTips, creating, 97-98
touch computing, evolution of, 7
Track, bounds, controlling for, 116
traditional circles, 63
transformations
  affine transformations, 66
data transformations, 32
  TypeConverters, 33-34
  value converters, 32-33
layout, 63-68
transition abstractions, building, 154-161
TransitionContainer control, creating, 154-161
TransitionPresenter control, 161
Transitions, 154
  applying, 159-161
  handling, 157-159
transparent PNGs, 309
TreeScope, enumeration values, 337
TreeWalker, 336
triggers (event), routed events, 245-246
TwirlEffect, parameterized pixel shaders, building, 231-235
two-pass approach, layout conversations, 51
type-based animations, 206-207
TypeConverters, data transformations, 33-34

U

UI Auditing, class handlers, registering, 249
UI automation, 331-332
  assemblies, 333
  automation APIs, 338-342
  automation elements, performing operations on, 342
  automation tree
    locating elements, 339-340
    navigating, 342
automation tree, navigating, 336-338
AutomationElement class, 333-335
AutomationPeer class, 333-335
custom controls, 343-349
events, listening to, 341-342
namespaces, 333
object model, 332-338
properties, 335-336
property values, reading, 341
resources, 349
UI controls, helper objects, attached properties, 103
UI virtualization, 130-131
3D virtualization, 140-142
classes, 131-132
component interaction, 132-133
containers, 130-131
recycling, 140
defered scrolling, 139-140
ItemContainerGenerator, 133-135
viewports, 130-131
virtualized controls, creating, 135-139
UIElement class, 5, 15-16, 50
UIs (user interfaces), 2-3
commands, 255-258
components, 257
ICommandSource interface, 262-266
request requery, 261-262
routed commands, 259
deferring operations, Dispatcher, 287
routed events, 243-245
attached events, 246-248
class handlers, 249-250
event triggers, 245-246
property mirrors, 246
weak events, 250-255
visual tree, 332
UISpy, 341
UndockFirstChild() method, 150, 152
undocking controls, 149-154
UniformGrid panel
layout, 50
properties, 71
unit testing, 111
UpdateScrollInfo method, 121
UseHover attached property, building, 95-100
user experience, accounting for, 8-9
user input, 6, 8
user-requested scrolling, responding to, 116
UserControl class, 22-23, 29
UX guidelines, visual design, 308

V
value converters, data transformations, 32-33
VanishingPointPanel, 56-58
vector graphics, designing, Adobe Illustrator, 309
vertex shaders, pixel shaders, compared, 223
vertical ScrollBars, 79
vertical scrolling, user-requested scrolling, responding to, 116
vertices, mesh surfaces, 219
Virtual Earth, 130
virtualization, 129-130, 142. See also UI virtualization
virtualization performance, 322-323
Virtual Earth, 130
Visual class, 13-16, 29
visual classes, 11
Adorner, 29
Adorner class, 24
Brush, 29
Brush-related classes, 25-27
ContentControl, 29
ContentControl class, 20
ContentPresenter, 29
ContentPresenter class, 20-21
Control, 29
Control class, 19-20
ControlTemplate class, 27
DataTemplate, 29
DataTemplate class, 27
Decorator, 29
Decorator class, 24
DependencyObject, 29
DependencyObject class, 12, 16
DispatcherObject, 29
DispatcherObject class, 12, 16
DrawingVisual, 29
DrawingVisual class, 13-15
FlowDocument, 18-19, 29
FrameworkElement, 29
FrameworkElement class, 15-16
GlyphRuns, 18-19, 29
hierarchy, 11-12
Image, 29
Image class, 25

How can we make this index more useful? Email us at indexes@samspublishing.com
visual classes

InkCanvas, 29
InkCanvas class, 28-29
ItemsControl, 29
ItemsControl class, 21-22
ItemsPanelTemplate, 29
ItemsPanelTemplate class, 27
ItemsPresenter, 29
MediaElement, 29
MediaElement class, 28
Panel, 29
Panel class, 23-24
Shape, 29
Shape class, 16-18
TextBlock, 18-19, 29
UserControl, 29
UserControl class, 22-23
Viewport3D, 29
Viewport3D class, 27-28
Visual, 29
Visual class, 13-16
visual children, custom panels, 52-55
visual design
Adobe Illustrator, vector manipulation, 309
clip geometries, 310-312
gradient brushes, 306-307
icons, nonstandard fonts, 308-309
opacity masks, 310
Photoshop, importing from, 309
relative transforms, 306-307
tile brushes, 304-305
transparent PNGs, 309
Windows Vista guidelines, 308
XAML coding conventions, 307-308
visual effects, 166
don shadow, implementing, 163
gloss effects, implementing, 164-165
opacity masks, implementing, 164
reflection effect, implementing, 161-162
transitions
applying, 159, 161
creating, 154-157
handling, 157-159
Visual Profiler, 327-328
visual richness, 319
Visual Studio, 312
visual styles, focus, 268
visual tree (UI), 332
VisualBrush class, 25-26
VisualChildrenCount property, 53
visuals, 318-319
VisualTreeHelper class, controls, customizing,
82-83

W–Z
watermarked text boxes, 269
Weak Delegates, 255
weak events, 250-255
delivering, IWeakEventListener, 254-255
implementing, 251-252
WEM (WeakEventManager), 251-254
Webdings font, icons, using for, 308-309
WeightedPanel, 58-63
WEM (WeakEventManager), 251-254
windows, 5
Windows Forms, 3
Windows Presentation Foundation (WPF), 1-3
Windows Vista guidelines, visual design, 308
worker threads, posting messages from,
287-289
world units, 186
WPF (Windows Presentation Foundation), 1-3
WPF Application Quality guide, 349
WrapPanel, properties, 71
WrapPanel class, 23
XAML (Extensible Application Markup Language)
3D world rendered output, 188
coding conventions, 307-308
xShared Boolean attribute, resources, cloning,
299-300
Unleashed takes you beyond the basics, providing an exhaustive, technically sophisticated reference for professionals who need to exploit a technology to its fullest potential. It’s the best resource for practical advice from the experts, and the most in-depth coverage of the latest technologies.
Your purchase of *WPF Control Development Unleashed* includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Sams book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, and Que.

**SAFARI BOOKS ONLINE** allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

**Activate your FREE Online Edition at**

[www.informit.com/safarifree](http://www.informit.com/safarifree)

**STEP 1:** Enter the coupon code: YTKFREH.

**STEP 2:** New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com