WPF to suddenly scale down your application to "normal" size. Similarly, if you're using a laptop with a high-resolution display, you probably expect to have slightly smaller windows—it's the price you pay to fit all your information onto a smaller screen. Furthermore, different users have different preferences. Some want richer detail, while others prefer to cram in more content.

So, how does WPF determine how big an application window *should* be? The short answer is that WPF uses the system DPI setting when it calculates sizes. But to understand how this really works, it helps to take a closer look at the WPF measurement system.

WPF Units

A WPF window and all the elements inside it are measured using *device-independent units*. A single device-independent unit is defined as 1/96 of an inch. To understand what this means in practice, you'll need to consider an example.

Imagine that you create a small button in WPF that's 96 by 96 units in size. If you're using the standard Windows DPI setting (96 dpi), each device-independent unit corresponds to one real, physical pixel. That's because WPF uses this calculation:

```
[Physical Unit Size] = [Device-Independent Unit Size] × [System DPI]
                     = 1/96 inch × 96 dpi
                     = 1 pixel
```

Essentially, WPF assumes it takes 96 pixels to make an inch because Windows tells it that through the system DPI setting. However, the reality depends on your display device.

For example, consider a 19-inch LCD monitor with a maximum resolution of 1600 by 1200 pixels. Using a dash of Pythagoras, you can calculate the pixel density for this monitor, as shown here:

$$[\text{Screen DPI}] = \frac{\sqrt{1600^2 + 1200^2}\ \text{pixels}}{19\ \text{inches}}$$

$$= 100\ \text{dpi}$$

In this case, the pixel density works out to 100 dpi, which is slightly higher than what Windows assumes. As a result, on this monitor a 96-by-96-pixel button will be slightly smaller than 1 inch.

On the other hand, consider a 15-inch LCD monitor with a resolution of 1024 by 768. Here, the pixel density drops to about 85 dpi, so the 96-by-96 pixel button appears slightly *larger* than 1 inch.

In both these cases, if you reduce the screen size (say, by switching to 800 by 600 resolution), the button (and every other screen element) will appear proportionately larger. That's because the system DPI setting remains at 96 dpi. In other words, Windows continues to assume it takes 96 pixels to make an inch, even though at a lower resolution it takes far fewer pixels.

■ **Tip** As you no doubt know, LCD monitors are designed to work best at a specific resolution, which is called the *native resolution*. If you lower the resolution, the monitor must use interpolation to fill in the extra pixels, which can cause blurriness. To get the best display, it's always best to use the native resolution. If you want larger windows, buttons, and text, consider modifying the system DPI setting instead (as described next).

System DPI

So far, the WPF button example works exactly the same as any other user interface element in any other type of Windows application. The difference is the result if you change the system DPI setting. In the previous generation of Windows, this feature was sometimes called *large fonts*. That's because the system DPI affects the system font size but often leaves other details unchanged.

■ **Note** Many Windows applications don't fully support higher DPI settings. At worst, increasing the system DPI can result in windows that have some content that's scaled up and other content that isn't, which can lead to obscured content and even unusable windows.

This is where WPF is different. WPF respects the system DPI setting natively and effortlessly. For example, if you change the system DPI setting to 120 dpi (a common choice for users of large high-resolution screens), WPF assumes that it needs 120 pixels to fill an inch of space. WPF uses the following calculation to figure out how it should translate its logical units to physical device pixels:

```
[Physical Unit Size] = [Device-Independent Unit Size] × [System DPI]
                     = 1/96 inch × 120 dpi
                     = 1.25 pixels
```

In other words, when you set the system DPI to 120 dpi, the WPF rendering engine assumes one device-independent unit equals 1.25 pixels. If you show a 96-by-96 button, the physical size will actually be 120 by 120 pixels (because 96 × 1.25 = 120). This is the result you expect—a button that's 1 inch on a standard monitor remains 1 inch in size on a monitor with a higher pixel density.

This automatic scaling wouldn't help much if it applied only to buttons. But WPF uses device-independent units for everything it displays, including shapes, controls, text, and any other ingredient you put in a window. As a result, you can change the system DPI to whatever you want, and WPF will adjust the size of your application seamlessly.

■ **Note** Depending on the system DPI, the calculated pixel size may be a fractional value. You might assume that WPF simply rounds off your measurements to the nearest pixel. However, by default, WPF does something different. If an edge of an element falls between pixels, it uses anti-aliasing to blend that edge into the adjacent pixels. This might seem like an odd choice, but it actually makes a fair bit of sense. Your controls won't necessarily have straight, clearly defined edges if you use custom-drawn graphics to skin them; so some level of anti-aliasing is already necessary.

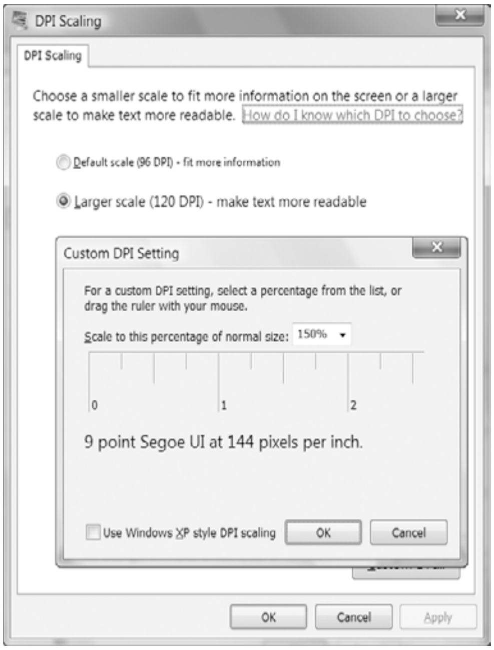
The steps for adjusting the system DPI depend on the operating system. The following sections explain what to do, depending on your operating system.

Windows XP

1. Right-click your desktop and choose Display.
2. Choose the Settings tab and click Advanced.
3. On the General tab, choose Normal Size (96 dpi) or Large Size (120 dpi). These are the two recommended options for Windows XP, because custom DPI settings are less likely to be supported by older programs. To try a custom DPI setting, choose Custom Setting. You can then specify a specific percentage value. (For example, 175% scales the standard 96 dpi to 168 dpi.)

Windows Vista

1. Right-click your desktop and choose Personalize.
2. In the list of links on the left, choose Adjust Font Size (DPI).
3. Choose between 96 or 120 dpi. Or click Custom DPI to use a custom DPI setting. You can then specify a percentage value, as shown in Figure 1-1. (For example, 175% scales the standard 96 dpi to 168 dpi.) In addition, when using a custom DPI setting, you have an option named Use Windows XP Style DPI Scaling, which is described in the sidebar "DPI Scaling with Windows Vista and Windows 7."



Choose a smaller scale to fit more information on the screen or a larger scale to make text more readable.

Default scale (96 DPI) - fit more information

[✓ Larger scale (120 DPI) - make text more readable]

For a custom DPI setting, select a percentage from the list, or drag the ruler with your mouse.

Scale to this percentage of normal size: 150%

9 point Segoe UI at 144 pixels per inch.

Use Windows XP style DPI scaling

Figure 1-1. *Changing the system DPI*

Windows 7

1. Right-click your desktop and choose Personalize.
2. In the list of links at the bottom-left of the window, choose Display.
3. Choose between Smaller (the default option), Medium, or Larger. Although these options are described by scaling percentages (100%, 125%, or 150%), they actually correspond to the DPI values 96, 120, and 144. You'll notice that the first two are the same standards found in Windows Vista and Windows XP, while the third one is larger still. Alternatively, you can click Set Custom Text Size to use a custom DPI percentage, as shown in Figure 1-1. (For example, 175% scales the standard 96 dpi to 168 dpi.) When using a custom DPI setting, you have an option named Use Windows XP Style DPI Scaling, which is described in the sidebar "DPI Scaling with Windows Vista and Windows 7."

DPI Scaling with Windows Vista and Windows 7

Because older applications are notoriously lacking in their support for high DPI settings, Windows Vista introduced a new technique called *bitmap scaling*. Windows 7 also supports this feature.

With bitmap scaling, when you run an application that doesn't appear to support high DPI settings, Windows resizes it as though it were an image. The advantage of this approach is that the application still believes it's running at the standard 96 dpi. Windows seamlessly translates input (such as mouse clicks) and routes them to the right place in the application's "real" coordinate system.

The scaling algorithm that Windows uses is a fairly good one—it respects pixel boundaries to avoid blurry edges and uses the video card hardware where possible to increase speed—but it inevitably leads to a fuzzier display. It also has a serious limitation in that Windows can't recognize older applications that *do* support high DPI settings. That's because applications need to include a manifest or call SetProcessDPIAware (in User32) to advertise their high DPI support. Although WPF applications handle this step correctly, applications created prior to Windows Vista won't use either approach and will be stuck with bitmap scaling even when they support higher DPIs.

There are two possible solutions. If you have a few specific applications that support high DPI settings but don't indicate it, you can configure that detail manually. To do so, right-click the shortcut that starts the application (in the Start menu) and choose Properties. On the Compatibility tab, enable the option named Disable Display Scaling on High DPI Settings. If you have a lot of applications to configure, this gets tiring fast.

The other possible solution is to disable bitmap scaling altogether. To do so, choose the Use Windows XP Style DPI Scaling option in the Custom DPI Setting dialog box shown in Figure 1-1. The only limitation of this approach is that there may be some applications that won't display properly (and possibly won't be usable) at high DPI settings. By default, Use Windows XP Style DPI Scaling is checked for DPI sizes of 120 or less but unchecked for DPI sizes that are greater.

Bitmap and Vector Graphics

When you work with ordinary controls, you can take WPF's resolution independence for granted. WPF takes care of making sure that everything has the right size automatically. However, if you plan to incorporate images into your application, you can't be quite as casual. For example, in traditional Windows applications, developers use tiny bitmaps for toolbar commands. In a WPF application, this approach is not ideal because the bitmap may display artifacts (becoming blurry) as it's scaled up or down according to the system DPI. Instead, when designing a WPF user interface, even the smallest icon is generally implemented as a vector graphic. *Vector graphics* are defined as a set of shapes, and as such they can be easily scaled to any size.

■ **Note** Of course, drawing a vector graphic takes more time than painting a basic bitmap, but WPF includes optimizations that are designed to lessen the overhead to ensure that drawing performance is always reasonable.

It's difficult to overestimate the importance of resolution independence. At first glance, it seems like a straightforward, elegant solution to a time-honored problem (which it is). However, in order to design interfaces that are fully scalable, developers need to embrace a new way of thinking.

The Architecture of WPF

WPF uses a multilayered architecture. At the top, your application interacts with a high-level set of services that are completely written in managed C# code. The actual work of translating .NET objects into Direct3D textures and triangles happens behind the scenes, using a lower-level unmanaged component called milcore.dll. milcore.dll is implemented in unmanaged code because it needs tight integration with Direct3D and because it's extremely performance-sensitive.

Figure 1-2 shows the layers at work in a WPF application.

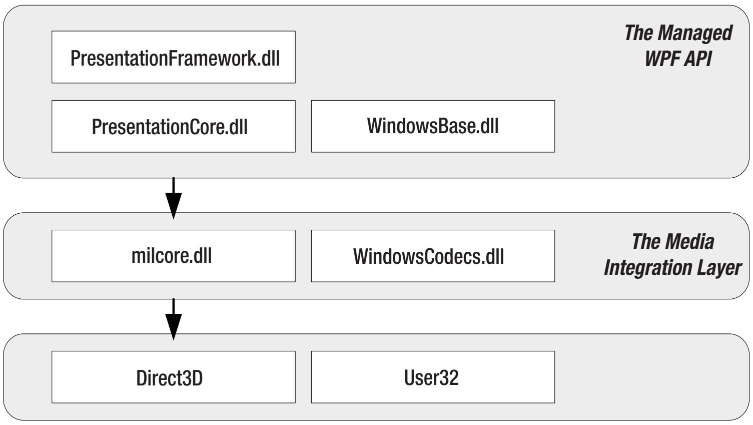


Figure 1-2. *The architecture of WPF*

Figure 1-2 includes these key components:

- **PresentationFramework.dll.** This holds the top-level WPF types, including those that represent windows, panels, and other types of controls. It also implements higher-level programming abstractions such as styles. Most of the classes you'll use directly come from this assembly.
- **PresentationCore.dll.** This holds base types, such as UIElement and Visual, from which all shapes and controls derive. If you don't need the full window and control abstraction layer, you can drop down to this level and still take advantage of WPF's rendering engine.
- **WindowsBase.dll.** This holds even more basic ingredients that have the potential to be reused outside of WPF, such as DispatcherObject and DependencyObject, which introduces the plumbing for dependency properties (a topic you'll explore in detail in Chapter 4).
- **milcore.dll.** This is the core of the WPF rendering system and the foundation of the Media Integration Layer (MIL). Its composition engine translates visual elements into the triangle and textures that Direct3D expects. Although milcore.dll is considered part of WPF, it's also an essential system component for Windows Vista and Windows 7. In fact, the Desktop Window Manager (DWM) uses milcore.dll to render the desktop.

■ **Note** milcore.dll is sometimes referred to as the engine for "managed graphics." Much as the common language runtime (CLR) manages the lifetime of a .NET application, milcore.dll manages the display state. And just as the CLR saves you from worrying about releasing objects and reclaiming memory, milcore.dll saves you from thinking about invalidating and repainting a window. You simply create the objects with the content you want to show, and milcore.dll paints the appropriate portions of the window as it is dragged around, covered and uncovered, minimized and restored, and so on.

- **WindowsCodecs.dll.** This is a low-level API that provides imaging support (for example, processing, displaying, and scaling bitmaps and JPEGs).
- **Direct3D.** This is the low-level API through which all the graphics in a WPF application are rendered.
- **User32.** This is used to determine what program gets what real estate. As a result, it's still involved in WPF, but it plays no part in rendering common controls.

The most important fact that you should realize is Direct3D renders *all* the drawing in WPF. It doesn't matter whether you have a modest video card or a much more powerful one, whether you're using basic controls or drawing more complex content, or whether you're running your application on Windows XP, Windows Vista, or Windows 7. Even two-dimensional shapes and ordinary text are transformed into triangles and passed through the 3-D pipeline. There is no fallback to GDI+ or User32.

The Class Hierarchy

Throughout this book, you'll spend most of your time exploring the WPF namespaces and classes. But before you begin, it's helpful to take a first look at the hierarchy of classes that leads to the basic set of WPF controls.

Figure 1-3 shows a basic overview with some of the key branches of the class hierarchy. As you continue through this book, you'll dig into these classes (and their relatives) in more detail.

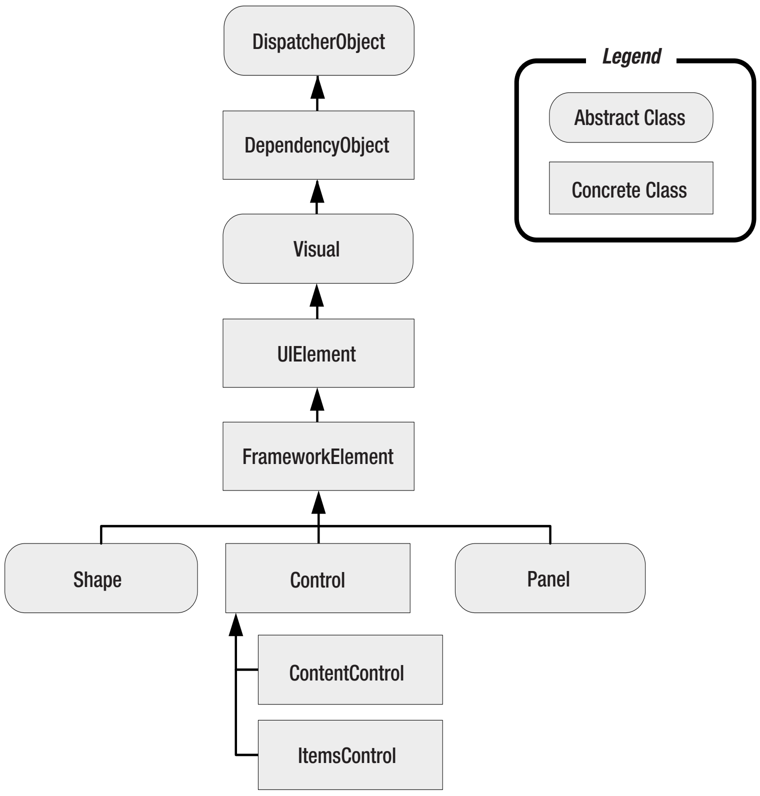


Figure 1-3. *The fundamental classes of WPF*

The following sections describe the core classes in this diagram. Many of these classes lead to whole branches of elements (such as shapes, panels, and controls).

■ **Note** The core WPF namespaces begin with System.Windows (for example, System.Windows, System.Windows.Controls, and System.Windows.Media). The sole exception is namespaces that begin with System.Windows.Forms, which are part of the Windows Forms toolkit.

System.Threading.DispatcherObject

WPF applications use the familiar single-thread affinity (STA) model, which means the entire user interface is owned by a single thread. It's not safe to interact with user interface elements from another thread. To facilitate this model, each WPF application is governed by a *dispatcher* that coordinates messages (which result from keyboard input, mouse movements, and framework processes such as layout). By deriving from DispatcherObject, every element in your user interface can verify whether code is running on the correct thread and access the dispatcher to marshal code to the user interface thread. You'll learn more about the WPF threading model in Chapter 31.

System.Windows.DependencyObject

In WPF, the central way of interacting with onscreen elements is through properties. Early on in the design cycle, the WPF architects decided to create a more powerful property model that baked in features such as change notification, inherited default values, and more economical property storage. The ultimate result is the *dependency property* feature, which you'll explore in Chapter 4. By deriving from DependencyObject, WPF classes get support for dependency properties.

System.Windows.Media.Visual

Every element that appears in a WPF window is, at heart, a Visual. You can think of the Visual class as a single drawing object that encapsulates drawing instructions, additional details about how the drawing should be performed (such as clipping, opacity, and transformation settings), and basic functionality (such as hit testing). The Visual class also provides the link between the managed WPF libraries and the milcore.dll that renders your display. Any class that derives from Visual has the ability to be displayed on a window. If you prefer to create your user interface using a lightweight API that doesn't have the higher-level framework features of WPF, you can program directly with Visual objects, as described in Chapter 14.

System.Windows.UIElement

UIElement adds support for WPF essentials such as layout, input, focus, and events (which the WPF team refers to by the acronym *LIFE*). For example, it's here that the two-step measure and arrange layout process is defined, which you'll learn about in Chapter 18. It's also here that raw mouse clicks and key presses are transformed to more useful events such as MouseEnter. As with properties, WPF implements an enhanced event-passing system called *routed events*. You'll learn how it works in Chapter 5. Finally, UIElement adds supports for commands (Chapter 9).

System.Windows.FrameworkElement

FrameworkElement is the final stop in the core WPF inheritance tree. It implements some of the members that are merely defined by UIElement. For example, UIElement sets the foundation for the WPF layout system, but FrameworkElement includes the key properties (such as HorizontalAlignment and Margin) that support it. UIElement also adds support for data binding, animation, and styles, all of which are core features.

System.Windows.Shapes.Shape

Basic shapes classes, such as Rectangle, Polygon, Ellipse, Line, and Path, derive from this class. These shapes can be used alongside more traditional Windows widgets such as buttons and text boxes. You'll start building shapes in Chapter 12.

System.Windows.Controls.Control

A *control* is an element that can interact with the user. It obviously includes classes such as TextBox, Button, and ListBox. The Control class adds additional properties for setting the font and the foreground and background colors. But the most interesting detail it provides is template support, which allows you to replace the standard appearance of a control with your own stylish drawing. You'll learn about control templates in Chapter 17.

■ **Note** In Windows Forms programming, every visual item in a form is referred to as a *control*. In WPF, this isn't the case. Visual items are called *elements*, and only some elements are actually controls (those that can receive focus and interact with the user). To make this system even more confusing, many elements are defined in the System.Windows.Controls namespace, even though they don't derive from System.Windows.Controls.Control and aren't considered controls. One example is the Panel class.

System.Windows.Controls.ContentControl

This is the base class for all controls that have a single piece of content. This includes everything from the humble Label to the Window. The most impressive part of this model (which is described in more detail in Chapter 6) is the fact that this single piece of content can be anything from an ordinary string to a layout panel with a combination of other shapes and controls.

System.Windows.Controls.ItemsControl

This is the base class for all controls that show a collection of items, such as the ListBox and TreeView. List controls are remarkably flexible—for example, using the features that are built into the ItemsControl class, you can transform the lowly ListBox into a list of radio buttons, a list of check boxes, a tiled display of images, or a combination of completely different elements that you've chosen. In fact, in WPF, menus, toolbars, and status bars are actually specialized lists, and the classes that implement them all derive from ItemsControl. You'll start using lists in Chapter 19 when you consider data binding. You'll learn to enhance them in Chapter 20, and you'll consider the most specialized list controls in Chapter 22.

System.Windows.Controls.Panel

This is the base class for all layout containers—elements that can contain one or more children and arrange them according to specific layout rules. These containers are the foundation of the WPF layout system, and using them is the key to arranging your content in the most attractive, flexible way possible. Chapter 3 explores the WPF layout system in more detail.

WPF 4

WPF is a relatively new technology. It's been part of several releases of .NET, with steady enhancements along the way:

- **WPF 3.0.** The first version of WPF was released with two other new technologies: Windows Communication Foundation (WCF) and Windows Workflow Foundation (WF). Together, these three technologies were called the .NET Framework 3.0.
- **WPF 3.5.** A year later, a new version of WPF was released as part of the .NET Framework 3.5. The new features in WPF are mostly minor refinements, including bug fixes and performance improvements.
- **WPF 3.5 SP1.** When the .NET Framework Service Pack 1 (SP1) was released, the designers of WPF had a chance to slip in a few new features, such as slick graphical effects (courtesy of pixel shaders) and the sophisticated DataGrid control.
- **WPF 4.** The latest of release of WPF adds a number of refinements, including some valuable new features that build on the existing WPF infrastructure. Some of the most notable changes include better text rendering, more natural animation, and support for Windows 7 features such as multitouch and the new taskbar.

New Features

This book covers all the concepts of WPF, including its snazziest new features and the core principles that haven't changed since its inception. However, if you're an experienced WPF developer, look for the "What's New" boxes that follow the introduction in most chapters. They detail content that's relatively new—in other words, features that appeared in WPF 3.5 SP1 or WPF 4. If you don't see a "What's New" box, it's a safe bet that the chapter deals with long-established WPF features that haven't changed in the latest release.

You can also use the following list to identify some of the most notable changes since WPF 3.0 and to find the chapters in this book where each feature is discussed:

- **More controls.** The family of WPF elements keeps growing. It now includes a professional DataGrid (Chapter 22), a standard DatePicker and Calendar (Chapter 6), and a native WebBrowser for HTML viewing and web surfing (Chapter 24). A separate download also adds the useful Ribbon control (Chapter 25), which can give any application a slick, modern look.
- **2-D drawing improvements.** Now the visual appearance of any element can be radically altered with PhotoShop-style effects through pixel shaders (using up to version 3 of the pixel-shader standard). Developers who want to manipulate individual pixels by hand can also generate and modify images with the WriteableBitmap class. Both features are described in Chapter 14.

- **Animation easing.** These functions allow you to create more lifelike animations that bounce, accelerate, and oscillate naturally. Chapter 15 has the full story.
- **Visual state manager.** First introduced in Silverlight, the visual state manager (Chapter 17) gives you an easier way to reskin controls without needing to understand the intricate details of their inner workings.
- **Windows 7.** Microsoft's newest operating system adds a slew of new features. WPF includes native support for the revamped taskbar, allowing you to use jump lists, icon overlays, progress notifications, and thumbnail toolbars (all of which are described in Chapter 23). And if you have the right hardware, you can use WPF's support for Windows 7 *multitouch* (Chapter 5), which is the ability to gesture on a touchscreen to manipulate visual objects.
- **Better rendering.** WPF continues to improve display quality and deal with the idiosyncrasies and scaling artifacts that can occur because of its resolution-independent drawing model. In WPF 4, you can use layout rounding to make sure layout containers line up with real pixel positions, guaranteeing a clear display (see Chapter 3). You can also do the same for rendered text, making sure it stays sharp even at vanishingly small sizes (Chapter 6).
- **Bitmap caching.** In the right scenario, you can spare the CPU's workload by caching complex vector art in video card memory. This technique is particularly handy when using animation, and it's described in Chapter 16.
- **XAML 2009.** WPF introduces a new version of the XAML markup standard that's used to declare the user interface in a window or page. It introduces a number of small refinements, but you probably won't use them just yet, because the standard isn't built into the WPF XAML compiler. Chapter 2 has more about the situation.

The WPF Toolkit

Before a new control makes its way into the WPF libraries of the .NET platform, it often begins in a separate Microsoft download known as the WPF Toolkit. But the WPF Toolkit isn't just a place to preview the future direction of WPF—it's also a great source of practical components and controls that are made available outside the normal WPF release cycle. For example, WPF doesn't include any sort of charting tools, but the WPF Toolkit includes a set of controls for creating bar, pie, bubble, scatter, and line graphs.

This book occasionally references the WPF Toolkit to point out a useful piece of functionality that's not available in the core .NET runtime. To download the WPF Toolkit, review its code, or read its documentation, surf to `http://wpf.codeplex.com`. There, you'll also find links to other Microsoft-managed WPF projects, including WPF Futures (which provides more experimental WPF features) and WPF testing tools.

Visual Studio 2010

Although you can craft WPF user interfaces by hand or using the graphic-design-oriented tool Expression Blend, most developers will start in Visual Studio and spend most (or all) of their time there. This book assumes you're using Visual Studio and occasionally explains how to use the Visual Studio interface to perform an important task, such as adding a resource, configuring project properties, or creating a control library assembly. However, you won't spend much time exploring Visual Studio's design-time frills. Instead, you'll focus on the underlying markup and code you need to create professional applications.

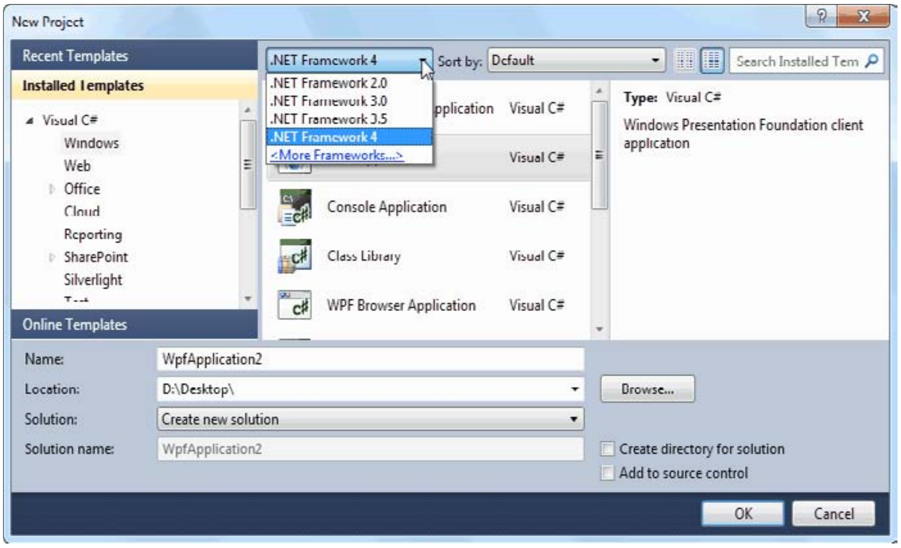
■ **Note** You probably already know how to create a WPF project in Visual Studio, but here's a quick recap. First, select File ➤ New ➤ Project. Then, pick the Visual C# ➤ Windows group (in the tree on the left), and choose the WPF Application template (in the list on the right). You'll learn about the more specialized WPF Browser Application template in Chapter 24. Once you pick a directory, enter a project name, and click OK, you'll end up with the basic skeleton of a WPF application.

Multitargeting

In the past, each version of Visual Studio was tightly coupled to a specific version of .NET. Visual Studio 2010 doesn't have this restriction—it allows you to design an application that targets any version of .NET from 2.0 to 4.

Although it's obviously not possible to create a WPF application with .NET 2.0, both .NET 3.0 and .NET 3.5 have WPF support. You may choose to target .NET 3.0 for the broadest possible compatibility (because .NET 3.0 applications can run on the .NET 3.0, 3.5, and 4 runtimes). Or, you may choose to target .NET 3.5 or 4 to get access to newer features in WPF or in the .NET platform.

When you create a new project in Visual Studio, you can choose the version of the .NET Framework that you're targeting from a drop-down list at the top of the New Project dialog box, just above the list of project templates (see Figure 1-4).


New Project

.NET Framework 4 | Sort by: Default

Type: Visual C#
Windows Presentation Foundation client application

Name: WpfApplication2
Location: D:\Desktop\
Solution: Create new solution
Solution name: WpfApplication2

Figure 1-4. Choosing the target version of the .NET Framework

You can also change the version you're targeting at any point afterward by double-clicking the Properties node in the Solution Explorer and changing the selection in the Target Framework list.

To provide accurate multitargeting, Visual Studio 2010 includes *reference assemblies* for each version of .NET. These assemblies include the metadata of every type but none of the code that's required to implement it. That means Visual Studio 2010 can use the reference assembly to tailor its IntelliSense and error checking, ensuring that you aren't able to use controls, classes, or members that aren't available in the version of .NET that you're targeting. It also uses this metadata to determine what controls should appear in the Toolbox, what members should appear in the Properties window and Object Browser, and so on, ensuring that the entire IDE is limited to the version you've chosen.

The .NET Client Profile

Oddly enough, there are two ways to target WPF 4. Your first option is to build an application that requires a standard installation of the full .NET Framework 4. Your second option is to build an application that requires the .NET Framework 4 Client Profile.

The client profile is the subset of the .NET Framework that's required for rich client applications like WPF. It doesn't include server-side features such as ASP.NET, debuggers, developer tools, code compilers, and legacy features (such as Oracle database support). More importantly, the client is smaller—it requires a download that's about 30 MB, while the full .NET Framework 4 redistributable tops 100 MB. Of course, if your application targets the .NET Framework 4 Client Profile and the client has the full version of the .NET Framework, it will still run without a hitch.

The client profile concept was introduced with .NET 3.5 SP1. However, it still had a few quirks in that release that prevented it from being the go-to standard. Now, in .NET 4, Microsoft has fine-tuned the feature set that's included in the client profile, with the goal of making it the standard choice for every application. In Visual Studio 2010, most projects target the .NET Framework 4 Client Profile automatically. (That's what you get if you choose .NET Framework 4 in the New Project dialog box.) If you change the Target Framework setting in your project properties, you'll see a more detailed list that has separate options for the .NET Framework 4 (the full version) and .NET Framework 4 Client Profile.

When choosing which version of .NET to target, it's often important to consider how widely the various runtimes are deployed. Ideally, your users should be able to run your WPF application without requiring another download and installation step. Here are a few guidelines that may help you decide:

- Windows Vista includes the .NET Framework 3.0.
- Windows 7 includes the .NET Framework 3.5 SP1.
- The .NET 4 Framework Client Profile is a recommended update (through Windows Update) for Windows Vista and Windows 7. It is an optional on Windows XP computers.

The Visual Studio Designer

Despite the fact that Visual Studio is *the* essential tool for WPF programming, previous versions have had a surprising gap in their abilities—they didn't provide a graphical designer for creating user interface. As a result, developers were forced to enter XAML markup by hand or switch between Visual Studio and the more design-oriented Expression Blend. Visual Studio 2010 finally corrects this oversight with a rich designer for creating WPF user interfaces.

But just because Visual Studio 2010 allows you to drag and drop WPF windows into existence doesn't mean you should start doing that right now—or at all. As you'll learn in Chapter 3, WPF uses a flexible and nuanced layout model that allows you to use different strategies for sizing and positioning the elements in your user interface. To get the result you need, you'll need to choose the right combination of layout containers, arrange them appropriately, and configure their properties. Visual Studio can help you out in this task, but it's far easier if you learn the basics of XAML markup and WPF layout *first*. Then, you'll be able to watch as Visual Studio's visual designer generates your markup, and you can modify it by hand as needed.

Once you've mastered the syntax of XAML (Chapter 2) and you've learned about the family of WPF layout controls (Chapter 3), it's up to you to choose how you want to create your windows. There are professional developers who use Visual Studio, those who use Expression Blend, those who write XAML by hand, and those who use a combination of both methods (for example, creating the basic layout structure by hand and then configuring it with the Visual Studio designer).

The Last Word

In this chapter, you took your first look at WPF and the promise it holds. You considered the underlying architecture and briefly considered the core classes.

WPF is the beginning of the future of Windows development. In time, it will become a system like User32 and GDI/GDI+, on top of which more enhancements and higher-level features are added. Eventually, WPF will allow you to design applications that would be impossible (or at least thoroughly impractical) using Windows Forms.

Clearly, WPF introduces many dramatic changes. However, there are five key principles that immediately stand out because they are so different from previous Windows user interface toolkits such as Windows Forms. These principles are the following:

- **Hardware acceleration.** All WPF drawing is performed through DirectX, which allows it to take advantage of the latest in modern video cards.
- **Resolution independence.** WPF is flexible enough to scale up or down to suit your monitor and display preferences, depending on the system DPI setting.
- **No fixed control appearance.** In traditional Windows development, there's a wide chasm between controls that can be tailored to suit your needs (which are known as *owner-drawn* controls) and those that are rendered by the operating system and essentially fixed in appearance. In WPF, everything from a basic Rectangle to a standard Button or more complex Toolbar is drawn using the same rendering engine and completely customizable. For this reason, WPF controls are often called *lookless controls*—they define the functionality of a control, but they don't have a hardwired "look."
- **Declarative user interfaces.** In the next chapter, you'll consider XAML, the markup standard you use to define WPF user interfaces. XAML allows you to build a window without using code. Impressively, XAML doesn't limit you to fixed, unchanging user interfaces. You can use tools such as data binding and triggers to automate basic user interface behavior (such as text boxes that update themselves when you page through a record source, or labels that glow when you hover overtop with the mouse), all without writing a single line of C#.

- **Object-based drawing.** Even if you plan to work at the lower-level visual layer (rather than the higher-level element layer), you won't work in terms of painting and pixels. Instead, you'll create shape objects and let WPF maintain the display in the most optimized manner possible.

You'll see these principles at work throughout this book. But before you go any further, it's time to learn about a complementary standard. The next chapter introduces XAML, the markup language used to define WPF user interfaces.

CHAPTER 2

■ ■ ■

XAML

XAML (short for Extensible Application Markup Language and pronounced as “zammel”) is a markup language used to instantiate .NET objects. Although XAML is a technology that can be applied to many different problem domains, its primary role in life is to construct WPF user interfaces. In other words, XAML documents define the arrangement of panels, buttons, and controls that make up the windows in a WPF application.

It’s unlikely that you’ll write XAML by hand. Instead, you’ll use a tool that generates the XAML you need. If you’re a graphic designer, that tool is likely to be a graphical design program such as Expression Blend. If you’re a developer, you’ll probably start with Visual Studio. Because both tools are equally at home with XAML, you can create a basic user interface with Visual Studio and then hand it off to a crack design team that can polish it up with custom graphics in Expression Blend. In fact, this ability to integrate the workflow between developers and designers is one of the key reasons that Microsoft created XAML.

In this chapter, you’ll get a detailed introduction to XAML. You’ll consider its purpose, its overall architecture, and its syntax. Once you understand the broad rules of XAML, you’ll know what is and isn’t possible in a WPF user interface—and how to make changes by hand when it’s necessary. More importantly, by exploring the tags in a WPF XAML document, you can learn a bit about the object model that underpins WPF user interfaces and get ready for the deeper exploration to come.

■ **What’s New** WPF 4 introduces XAML 2009, an updated version of XAML with a number of useful refinements. However, there’s a significant shortcoming: currently, it’s possible to use XAML 2009 only in loose XAML files. Although Visual Studio supports both loose and compiled XAML (as you’ll learn in this chapter), compiled XAML is the standard. Not only does it work with the code-behind model, allowing you to wire up code with a minimum of effort, it also ensures that your compiled application with be smaller and will load slightly faster. For all these reasons, you won’t use XAML 2009 with the examples in this book. However, you’ll get a preview of the enhancements in the section “XAML 2009.” This information will prepare you for future releases of WPF, because XAML 2009 is slated to become the new standard—once Microsoft has time to rewrite, test, and optimize WPF’s XAML compiler.

Understanding XAML

Developers realized long ago that the most efficient way to tackle complex, graphically rich applications is to separate the graphical portion from the underlying code. That way, artists can own the graphics, and developers can own the code. Both pieces can be designed and refined separately, without any versioning headaches.

Graphical User Interfaces Before WPF

With traditional display technologies, there's no easy way to separate the graphical content from the code. The key problem with a Windows Forms application is that every form you create is defined entirely in C# code. As you drop controls onto the design surface and configure them, Visual Studio quietly adjusts the code in the corresponding form class. Sadly, graphic designers don't have any tools that can work with C# code.

Instead, artists are forced to take their content and export it to a bitmap format. These bitmaps can then be used to skin windows, buttons, and other controls. This approach works well for straightforward interfaces that don't change much over time, but it's extremely limiting in other scenarios. Some of its problems include the following:

- Each graphical element (background, button, and so on) needs to be exported as a separate bitmap. That limits the ability to combine bitmaps and use dynamic effects such as anti-aliasing, transparency, and shadows.
- A fair bit of user interface logic needs to be embedded in the code by the developer. This includes button sizes, positioning, mouseover effects, and animations. The graphic designer can't control any of these details.
- There's no intrinsic connection between the different graphical elements, so it's easy to end up with an unmatched set of images. Tracking all these items adds complexity.
- Bitmaps can't be resized without compromising their quality. For that reason, a bitmap-based user interface is resolution-dependent. That means it can't accommodate large monitors and high-resolution displays, which is a major violation of the WPF design philosophy.

If you've ever been through the process of designing a Windows Forms application with custom graphics in a team setting, you've put up with a lot of frustration. Even if the interface is designed from scratch by a graphic designer, you'll need to re-create it with C# code. Usually, the graphic designer will simply prepare a mock-up that you need to translate painstakingly into your application.

WPF solves this problem with XAML. When designing a WPF application in Visual Studio, the window you're designing isn't translated into code. Instead, it's serialized into a set of XAML tags. When you run the application, these tags are used to generate the objects that compose the user interface.

■ **Note** It's important to understand that WPF doesn't require XAML. There's no reason Visual Studio couldn't use the Windows Forms approach and create code statements that construct your WPF windows. But if it did, your window would be locked into the Visual Studio environment and available to programmers only.

In other words, WPF doesn't require XAML. However, XAML opens up worlds of possibilities for collaboration, because other design tools understand the XAML format. For example, a savvy designer can use a tool such as Expression Design to fine-tune the graphics in your WPF application or a tool such as Expression Blend to build sophisticated animations for it. After you've finished this chapter, you may want to read a Microsoft white paper at `http://windowsclient.net/wpf/white-papers/thenewiteration.aspx` that reviews XAML and explores some of the ways developers and designers can collaborate on a WPF application.

■ **Tip** XAML plays the same role for Windows applications as control tags do for ASP.NET web applications. The difference is that the ASP.NET tagging syntax is designed to look like HTML, so designers can craft web pages using ordinary web design applications such as FrontPage and Dreamweaver. As with WPF, the actual code for an ASP.NET web page is usually placed in a separate file to facilitate this design.

The Variants of XAML

There are actually several different ways people use the term *XAML*. So far, I've used it to refer to the entire language of XAML, which is an all-purpose XML-based syntax for representing a tree of .NET objects. (These objects could be buttons and text boxes in a window or custom classes you've defined. In fact, XAML could even be used on other platforms to represent non-.NET objects.)

There are also several subsets of XAML:

- **WPF XAML** encompasses the elements that describe WPF content, such as vector graphics, controls, and documents. Currently, it's the most significant application of XAML, and it's the subset you'll explore in this book.
- **XPS XAML** is the part of WPF XAML that defines an XML representation for formatted electronic documents. It's been published as the separate XML Paper Specification (XPS) standard. You'll explore XPS in Chapter 28.
- **Silverlight XAML** is a subset of WPF XAML that's intended for Silverlight applications. Silverlight is a cross-platform browser plug-in that allows you to create rich web content with two-dimensional graphics, animation, and audio and video. Chapter 1 has more about Silverlight, or you can visit `http://silverlight.net` to learn about it in detail.
- **WF XAML** encompasses the elements that describe Windows Workflow Foundation (WF) content. You can learn more about WF at `http://tinyurl.com/4y4apd`.

XAML Compilation

The creators of WPF knew that XAML needed to not just solve the problem of design collaboration—it also needed to be fast. And though XML-based formats such as XAML are flexible and easily portable to other tools and platforms, they aren't always the most efficient option. XML was designed to be logical, readable, and straightforward, not compact.

WPF addresses this shortcoming with Binary Application Markup Language (BAML). BAML is really nothing more than a binary representation of XAML. When you compile a WPF application in Visual Studio, all your XAML files are converted into BAML, and that BAML is then embedded as a resource into the final DLL or EXE assembly. BAML is *tokenized*, which means lengthier bits of XAML are replaced with shorter tokens. Not only is BAML significantly smaller, but it's also optimized in a way that makes it faster to parse at runtime.

Most developers won't worry about the conversion of XAML to BAML because the compiler performs it behind the scenes. However, it is possible to use XAML without compiling it first. This might make sense in scenarios that require some of the user interface to be supplied just in time (for example, pulled out of a database as a block of XAML tags). You'll see how this works later in the section "Loading and Compiling XAML."

Creating XAML With Visual Studio

In this chapter, you'll take a look at all the details of XAML markup. Of course, when you're designing an application, you won't write all your XAML by hand. Instead, you'll use a tool such as Visual Studio that can drag and drop your user interface into existence. Based on that, you might wonder whether it's worth spending so much time studying the syntax of XAML.

The answer is a resounding *yes*. Understanding XAML is critical to WPF application design. It will help you learn key WPF concepts, such as attached properties (in this chapter), layout (Chapter 3), routed events (Chapter 4), the content model (Chapter 6), and so on. More importantly, there is a whole host of tasks that are possible—or are far easier to accomplish—only with handwritten XAML. They include the following:

- **Wiring up event handlers.** Attaching event handlers in the most common places—for example, to the Click event of a Button—is easy to do in Visual Studio. However, once you understand how events are wired up in XAML, you'll be able create more sophisticated connections. For example, you can set up an event handler that responds to the Click event of every button in a window. Chapter 5 has more about this technique.

Writing data binding expressions. Data binding allows you to extract data from an object and display it in a linked element. To set up this relationship and configure how it works, you must add a data binding expression to your XAML markup. Chapter 8 introduces data binding.

Defining resources. Resources are objects that you define once in your XAML and in a special section of your XAML and then reuse in various places in your markup. Resources allow you to centralize and standardize formatting and create nonvisual objects such as templates and animations. Chapter 10 shows how to create and use resources.

Defining animations. Animations are a common ingredient in XAML applications. Usually, they're defined as resources, constructed using XAML markup, and then linked to other controls (or triggered through code). Currently, Visual Studio has no design-time support for crafting animations. Chapter 15 delves into animation.

Defining control templates. WPF controls are designed to be *lookless*, which means you can substitute your custom visuals in place of the standard appearance. To do so, you must create your own control template, which is nothing more than a block of XAML markup. Chapter 17 tackles control templates.

Most WPF developers use a combination of techniques, laying out some of their user interface with a design tool (Visual Studio or Expression Blend) and then fine-tuning it by editing the XAML markup by hand. However, you'll probably find that it's easiest to write all your XAML by hand until you learn about layout containers in Chapter 3. That's because you need to use a layout container to properly arrange multiple controls in a window.

XAML Basics

The XAML standard is quite straightforward once you understand a few ground rules:

- Every element in a XAML document maps to an instance of a .NET class. The name of the element matches the name of the class *exactly*. For example, the element <Button> instructs WPF to create a Button object.
- As with any XML document, you can nest one element inside another. As you'll see, XAML gives every class the flexibility to decide how it handles this situation. However, nesting is usually a way to express *containment*—in other words, if you find a Button element inside a Grid element, your user interface probably includes a Grid that contains a Button inside.
- You can set the properties of each class through attributes. However, in some situations an attribute isn't powerful enough to handle the job. In these cases, you'll use nested tags with a special syntax.

■ **Tip** If you're completely new to XML, you'll probably find it easier to review the basics before you tackle XAML. To get up to speed quickly, try the free web-based tutorial at `http://www.w3schools.com/xml`.

Before continuing, take a look at this bare-bones XAML document, which represents a new blank window (as created by Visual Studio). The lines have been numbered for easy reference:

```
1  <Window x:Class="WindowsApplication1.Window1"
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4      Title="Window1" Height="300" Width="300">
5
6      <Grid>
7      </Grid>
8  </Window>
```