PART IV
Creating Graphical Output

15
.NET Compact Framework Graphics

This chapter introduces the basics of creating graphical output from .NET Compact Framework programs.

THIS CHAPTER DESCRIBES the support that the .NET Compact Framework provides programs for creating graphical output. As we mention elsewhere in this book, we prefer using .NET Compact Framework classes whenever possible. To accomplish something beyond what the .NET Compact Framework supports, however, we drill through the managed-code layer to the underlying Win32 API substrate. This chapter and the two that follow discuss the .NET Compact Framework's built-in support for creating graphical output; these chapters also touch on limitations of that support and how to supplement that support with the help of GDI functions.

An Introduction to .NET Compact Framework Graphics

In general, programs do not create graphical output by drawing directly to device hardware.[1] A program typically calls a library of graphical output functions. Those drawing functions, in turn, rely on device drivers that

1. But when necessary for performance reasons or to access device-specific features, a program might bypass the intervening software layers and interact with hardware.

provide the device-specific elements needed to create output on a device. Historically, creating output on a graphic device such as a display screen or a printer involves these software layers:

- Drawing program
- Graphic function library
- Graphic device driver (display driver or printer driver)

The core graphics library on desktop Windows is the Graphics Device Interface (GDI, `gdi32.dll`). With the coming of .NET, Microsoft added a second library (GDI+, `gdiplus.dll`[2]) to supplement GDI drawing support. This second library provides a set of enhancements on top of the core GDI drawing functions. While the primary role for GDI+ was to support graphics for the managed-code library, it also provides a nice bonus for native-mode application programmers: the library can be called from unmanaged (native-mode) C++ programs. On the desktop, these two graphic libraries—GDI and GDI+—provide the underpinnings for all of the .NET graphic classes. And so, with .NET Framework programs running on the Windows desktop, the architecture of graphical output involves the following elements:

- Managed-code program
- Shared managed-code library (`System.Drawing.dll`)
- GDI+ native-code library (`gdiplus.dll`)
- GDI native-code library (`gdi32.dll`)
- Graphic device driver (display driver or printer driver)

Windows CE supports a select set of GDI drawing functions. There is no library explicitly named GDI in Windows CE. Instead, the graphical output functions reside in the `coredll.dll` library. These functions are exactly like their desktop counterparts, so even if there is no library named GDI in Windows CE, we refer to these functions as GDI functions.

2. GDI+ is a native-mode, unmanaged-code library.

.NET Framework Drawing and Desktop Graphic Device Drivers

With the introduction of the .NET Framework, no changes were required to the graphic device drivers of any version of Microsoft Windows. That is, the device driver model used by both display screens and printer drivers was robust enough to support the .NET drawing classes.

Of the 400 or so functions that exist on desktop versions of GDI, only about 85 are included in Windows CE. Windows CE has none of the drawing functions from the extended desktop graphics library, GDI+. This places some limits on the extent to which Windows CE can support .NET drawing functions.

With just 85 of the graphical functions from the desktop's GDI library and none of the functions from GDI+, you might wonder whether Windows CE has enough graphics support to create interesting graphical output. The answer is a resounding: Yes! While there are not a lot of graphical functions, the ones that are present were hand-picked as the ones that programs tend to use most. For example, there is a good set of text, raster, and vector functions. A program can use fonts to create rich text output, display bitmaps along with other kinds of raster data (like JPEG files), and draw vector objects such as lines and polygons.

For graphical output, .NET Compact Framework programs rely on `System.Drawing.dll`, which is also the name of the graphical output library in the desktop .NET Framework. At 38K, the .NET Compact Framework library is significantly smaller than the 456K of its counterpart on the desktop. While the desktop library supports five namespaces, the .NET Compact Framework version supports one: `System.Drawing` (plus tiny fragments of two other namespaces). The architecture for drawing from a .NET Compact Framework program is as follows:

- Managed-code program
- Managed-code library (`System.Drawing.dll`)
- GDI functions in the native-code library (`coredll.dll`)
- Graphic device driver (display or printer)

From the arrangement of these software layers, a savvy .NET Compact Framework programmer can divine two interesting points: (1) The managed-code library depends on the built-in GDI drawing functions, and managed-code programs can do the same; and (2) as on the desktop, display screens and printers require a dedicated graphic driver to operate.

If Possible, Delegate Graphical Output to a Control

Before you dig into .NET Compact Framework graphics, ask yourself whether you want to create the graphical output yourself or can delegate that work to a control. If a control exists that can create the output you require, you can save yourself a lot of effort by using that control instead of writing the drawing code yourself. For example, the `PictureBox` control displays bitmaps and JPEG images with little effort. Aside from that single control, however, most controls are text-oriented.

Doing your own drawing—and making it look good—takes time and energy. By delegating graphical output to controls, you can concentrate on application-specific work. The built-in controls support a highly interactive, if somewhat text-oriented, user interface.

Sometimes, however, you do your own drawing to give your program a unique look and feel. In that case, you can create rich, graphical output by using classes in the .NET Compact Framework's `System.Drawing` namespace.

Drawing Surfaces

On the Windows desktop, there are four types of drawing surfaces:

1. Display screens
2. Printers
3. Bitmaps
4. Metafiles

When we use the term *drawing surface*, we mean either a physical drawing surface or a logical drawing surface. Two of the four drawing surfaces in the list are physical drawing surfaces, which require dedicated device drivers: display screens and printers. The other two drawing surfaces are logical drawing surfaces: bitmaps and metafiles. These latter two store pictures for eventual output to a device.

Bitmaps and metafiles are similar enough that they share a common base class in the desktop .NET Framework: the `Image`[3] class. Metafiles are not officially supported in Windows CE, however, and so their wrapper, the `Metafile`[4] class, does not exist in the current version of the .NET Compact Framework. Because metafiles might someday be supported in a future version of the .NET Compact Framework, they are worth a brief mention here.

Display Screens

The display screen plays a central role in all GUI environments because it is on the display screen that a user interacts with the various GUI applications. The real stars of the display screen are the windows after which the operating system gets its name. A window acts as a *virtual console*[5] for interacting with a user. The physical console for a desktop PC consists of a display screen, a mouse, and a keyboard. On a Pocket PC, the physical console is made up of a display screen, a stylus and a touch-sensitive screen for pointing, and hardware buttons for input (supported, of course, by the on-screen keyboard).

All graphical output on the display screen is directed to one window or another. Enforcement of window boundaries relies on *clipping*. Clipping is the establishment and enforcement of drawing boundaries; a program can draw inside clipping boundaries but not outside them. The simplest clipping boundaries are a rectangle. The area inside a window where a program may draw is referred to as the window's *client area*.

---

3. Fully qualified name: `System.Drawing.Image`.
4. Fully qualified name: `System.Drawing.Imaging.Metafile`.
5. A term we first heard from Marlin Eller, a member of the GDI team for Windows 1.x.

Printers

Printers are the best-established and most-connected peripherals in the world of computers. While some industry pundits still rant about the soon-to-arrive paperless office, just the opposite has occurred. Demand for printed output has continued to go up, not down. Perhaps the world of computers—with its flashing LCD displays, volatile RAM, and ever-shrinking silicon—makes a person want something that is more real.

Printing from Windows CE–powered devices is still in its infancy, which is a nice way to say that this part of the operating system is less feature-rich than other portions. Why is that? The official story is that there is not a good enough business case for adding better printing support, meaning that users have not asked for it. The fundamental question, then, is "Why haven't users asked for better printing for Windows CE?" Perhaps it is because users are used to printing from desktop PCs. Or perhaps the problem stems from the lack of printing support in programs bundled with Pocket PCs (like Pocket Word and Pocket Excel). Whatever the cause, we show you several ways to print in Chapter 17 so that you can decide whether the results are worth the effort.

Bitmaps

Bitmaps provide a way to store a picture. Like its desktop counterparts, Windows CE supports device-independent bitmaps (DIBs) as first-class citizens. In-memory bitmaps can be created of any size[6] and treated like any other drawing surface. After a program has drawn to a bitmap, that image can be put on the display screen.

If you look closely, you can see that Windows CE and the .NET Compact Framework support other raster formats. Supported formats include GIF, PNG, and JPEG. When Visual Studio .NET reads files with these formats (which it uses for inclusion in image lists, for example), it converts the raster data to a bitmap. The same occurs when a PNG or JPEG file is read from the object store into a .NET Compact Framework program. Whatever external format is used for raster data, Windows CE prefers bitmaps. In this chapter, we show how to create a bitmap from a variety of sources and

6. The amount of available system memory limits the bitmap size.

how to draw those bitmaps onto the display screen from a .NET Compact Framework program.

Compressed Raster Support on Custom Windows CE Platforms

Pocket PCs support the compressed raster formats, that is, GIF, PNG, and JPEG files. Custom Windows CE platforms must include the image decompression library, named `imgdecmp.dll`, to receive that same support.

Metafiles

A second picture-storing mechanism supported by desktop Windows consists of metafiles. A metafile is a record-and-playback mechanism that stores the details of GDI drawing calls. The 32-bit version of Windows metafiles are known as *Enhanced Metafiles* (EMFs). The following Win32 native metafile functions are exported from `coredll.dll` but are not officially supported in Windows CE, although they might gain official support in some future version of Windows CE:

- `CreateEnhMetaFile`
- `PlayEnhMetaFile`
- `CloseEnhMetaFile`
- `DeleteEnhMetaFile`

Supported Drawing Surfaces

Of these four types of drawing surfaces, three have official support in Windows CE: display screens, printers, and bitmaps. Only two are supported by the .NET Compact Framework: display screens and bitmaps. Support for bitmaps centers around the `Bitmap`[7] class, which we discuss later in this chapter. We start this discussion of graphical output with the drawing surface that is the focus in all GUI systems: the display screen.

7. Fully qualified name: `System.Drawing.Bitmap`.

Drawing Function Families

All of the graphical output functions can be organized into one of three drawing function families:

- Text
- Raster
- Vector

Each family has its own set of drawing attributes and its own logic for how its drawing is done. The distinction between these three kinds of output extends from the drawing program into the graphic device drivers. Each family is complex enough for a programmer to spend many years mastering the details and intricacies of each type of drawing. The drawing support is rich enough, however, so that you do not have to be an expert to take advantage of what is offered.

Text Output

For drawing text, the most important issue involves selection of the font because all text drawing requires a font, and the font choice has the greatest impact on the visual display of text. The only other drawing attribute that affects text drawing is color—both the foreground text and the color of the background area. We touch on text briefly in this chapter, but the topic is important enough to warrant a complete chapter, which we provide in Chapter 16.

Raster Output

Raster data involves working with arrays of pixels, sometimes known as bitmaps or image data. Internally, raster data is stored as a DIB. As we discuss in detail later in this chapter, six basic DIB formats are supported in the various versions of Windows: 1, 4, 8, 16, 24, and 32 bits per pixel. Windows CE adds a seventh DIB format to this set: 2 bits per pixel.

Windows CE provides very good support for raster data. You can dynamically create bitmaps, draw on bitmaps, display them for the user to see, and store them on disk. A bitmap, in fact, has the same rights and

privileges as the display screen. By this we mean that you use the same set of drawing functions both for the screen and for bitmaps. This means you can use bitmaps to achieve interesting effects by first drawing to a bitmap and subsequently copying that image to the display screen. An important difference from desktop versions of Windows is that Windows CE does not support any type of coordinate transformations, and in particular there is no support for the rotation of bitmaps; the .NET Compact Framework inherits these limitations because it relies on native Win32 API functions for all of its graphics support.

Vector Output

Vector drawing involves drawing geometric figures like ellipses, rectangles, and polygons. There are, in fact, two sets of drawing functions for each type of figure. One set draws the border of geometric figures with a *pen*. The other set of functions fill the interiors of geometric figures using a *brush*. You'll find more details on vector drawing later in this chapter.

.NET Compact Framework Graphics

The .NET Framework has six namespaces that support the various graphical output classes. In the .NET Compact Framework, just one namespace has made the cut: `System.Drawing`. This namespace and its various classes are packaged in the `System.Drawing.dll` assembly. For a detailed comparison between the graphics support in the .NET Framework and in the .NET Compact Framework, see the sidebar titled Comparing Supported Desktop and Smart-Device Drawing.

Comparing Supported Desktop and Smart-Device Drawing

The `System.Drawing` namespace in the .NET Compact Framework holds the primary elements used to draw on a device screen from managed code. The desktop .NET Framework provides five namespaces for creating graphical output, but in the .NET Compact Framework this has been pared

back to two: `System.Drawing` and `System.Drawing.Design` (plus some fragments from two other namespaces).

Table 15.1 summarizes the .NET namespaces supported in the desktop .NET Framework, along with details of how these features are supported in the .NET Compact Framework. The `System.Drawing` namespace supports drawing on a device screen. A second namespace, `System.Drawing.Design`, helps when building a custom control. In particular, this namespace contains elements used to support design-time drawing of controls (i.e., drawing controls while they are being laid out inside the Designer). The elements of this namespace reside in the `System.CF.Design.dll` assembly, a different name from the assembly name used for the desktop. The change in the file name makes it clear that this file supports .NET Compact Framework programming.

On the surface, it would be easy to conclude that Microsoft gutted the desktop `System.Drawing.dll` library in creating the .NET Compact Framework edition. For one thing, the desktop version is a whopping 456K, while the compact version is a scant 38K. What's more, the desktop version supports 159 classes, while the compact version has a mere 17 classes. A more specific example of the difference between the desktop .NET Framework and the .NET Compact Framework—from a drawing perspective—is best appreciated by examining the `Graphics` class (a member of the `System.Drawing` namespace). The desktop .NET Framework version of this class supports 244 methods and 18 properties; the .NET Compact Framework version supports only 26 methods and 2 properties. By this accounting, it appears that the prognosis of "gutted" is correct. Yet, as any thinking person knows, looks can be deceiving.

To understand better the difference between the desktop .NET Framework and the .NET Compact Framework, we have to dig deeper into the `Graphics` class. To really see the differences between the desktop and compact versions, we must study the overloaded methods. If we do, we see that the desktop .NET Framework provides many overloaded methods for each drawing call, while the .NET Compact Framework provides far

TABLE 15.1: Desktop .NET Framework Drawing Namespaces in the .NET Compact Framework

Namespace	Description	Support in the .NET Compact Framework
`System.Drawing`	Core drawing objects, data structures, and functions	A minimal set that allows for the drawing of text, raster, and vector objects with no built-in coordinate transformation
`System.Drawing.Design`	Support for the Designer and the various graphic editors of Visual Studio .NET	Support provided by a .NET Compact Framework–specific alternative library named `System.CF.Design.dll`
`System.Drawing.Drawing2D`	Support for advanced graphic features including blends, line caps, line joins, paths, coordinate transforms, and regions	Not supported in the .NET Compact Framework (except for the `CombineMode` enumeration)
`System.Drawing.Imaging`	Support for storage of pictures in metafiles and bitmaps; bitmap conversion; and management of metadata in image files	Not supported in the .NET Compact Framework (except for the `ImageAttributes` class)
`System.Drawing.Printing`	Rich support for printing and the user interface for printing	Not supported in the .NET Compact Framework
`System.Drawing.Text`	Font management	Not supported in the .NET Compact Framework

fewer. For example, the desktop .NET Framework provides six different ways to call `DrawString` (the text drawing function), while there is only one in the .NET Compact Framework. And there are 34 versions of `DrawImage` (the function for drawing a bitmap) but only four in the .NET Compact Framework.

We have, in short, fewer ways to draw objects—but in general we can draw most of the same things with the .NET Compact Framework that we can draw on the desktop. This supports a central design goal of Windows CE, which is to be a small, compact operating system. Win32 programmers who have worked in Windows CE will recognize that a similar trimming has been done to define the Windows CE support for the Win32 API. Instead of calling this a "subset," we prefer to take a cue from the music recording industry and use the term "greatest hits." The .NET Compact Framework implementation of the `System.Drawing` namespace is, we believe, the greatest hits of the desktop `System.Drawing` namespace.

In comparing the desktop .NET Framework to the .NET Compact Framework, an interesting pattern emerges that involves floating-point numbers. In the desktop .NET Framework, most of the overloaded methods take floating-point coordinates. For all of the overloaded versions of the `DrawString` methods, you can *only* use floating-point coordinates. In the .NET Compact Framework, few drawing functions have floating-point parameters—most take either `int32` or a `Rectangle` to specify drawing coordinates. A notable exception is the `DrawString` function, which never takes integer coordinates in the desktop .NET Framework; in the .NET Compact Framework, it is the sole drawing method that accepts floating-point values.

It is worth noting that the underlying drawing functions (both in the operating system and at the device driver level) exclusively use integer coordinates. The reason is more an accident of history than anything else. The Win32 API and its supporting operating systems trace their origins back to the late 1980s, when the majority of systems did not have built-in floating-point hardware. Such support is taken for granted today,

which is no doubt why the .NET Framework has such rich support for floating-point values.

A fundamental part of any graphics software is the coordinate system used to specify the location of objects drawn on a drawing surface. The desktop .NET Framework supports seven distinct drawing coordinate systems in the `GraphicsUnit` enumeration. Among the supported coordinates systems are `Pixel`, `Inch`, and `Millimeter`. While the .NET Compact Framework supports this same enumeration, it has only one member: `Pixel`. This means that when you draw on a device screen, you are limited to using pixel coordinates. One exception involves fonts, whose height is always specified in `Point` units.

This brings up another difference between the desktop .NET Framework and the .NET Compact Framework: available coordinate transformations. The desktop provides a rich set of coordinate transformations—scrolling, scaling, and rotating—through the `Matrix` class and the 3×3 geometric transform provided in the `System.Drawing.Drawing2D` namespace. The .NET Compact Framework, by contrast, supports no coordinate mapping. That means that, on handheld devices, application software that wants to scale, scroll, or rotate must handle the arithmetic itself because neither the .NET Compact Framework nor the underlying operating system provides any coordinate transformation helpers. What the .NET Compact Framework provides, as far as coordinates go, is actually the same thing that the underlying Windows CE system provides: pixels, more pixels, and only pixels.

While it might be lean, the set of drawing services provided in the .NET Compact Framework is surprisingly complete. That is, almost anything you can draw with the desktop .NET Framework can be drawn with the .NET Compact Framework. The key difference between the two implementations is that the desktop provides a far wider array of tools and helpers for drawing. Programmers of the desktop .NET Framework are likely to have little trouble getting comfortable in the .NET Compact Framework, once they get used to the fact that there are far fewer features. But those same programmers

are likely to be a bit frustrated when porting desktop .NET Framework code to the .NET Compact Framework world and are likely to have to rewrite and retrofit quite a few of their applications' drawing elements.

The Role of the `Graphics` Class

The most important class for creating graphical output is the `Graphics`[8] class. It is not the only class in the `System.Drawing` namespace, but only the `Graphics` class has drawing methods. This class holds methods like `DrawString` for drawing a string of text, `DrawImage` for displaying a bitmap onto the display screen,[9] and `DrawRectangle` for drawing the outline of a rectangle. Here is a list of the other classes in the `System.Drawing` namespace for the .NET Compact Framework:

- `Bitmap`
- `Brush`
- `Color`
- `Font`
- `FontFamily`
- `Icon`
- `Image`
- `Pen`
- `Region`
- `SolidBrush`
- `SystemColors`

These other classes support objects that aid in the creation of graphical output, but none has any methods that actually cause graphical output to appear anywhere. So while you are going to need these other classes and will use these other classes, they play a secondary role to the primary graphical output class in the .NET Compact Framework: `Graphics`.

8. Fully qualified name: `System.Drawing.Graphics`.
9. `DrawImage` can also be used to draw bitmaps onto other bitmaps.

Drawing Support for Text Output

Table 15.2 summarizes the methods of the `Graphics` class that support text drawing. The `DrawString` method draws text, while the `MeasureString` method calculates the bounding box of a text string. This calculation is needed because graphical output involves putting different types of graphical objects on a sea of pixels. When dealing with a lot of text, it is important to measure the size of each textbox to make sure that the spacing matches the spacing as defined by the font designer. Failure to use proper spacing creates a poor result. In the worst cases, it makes the output of your program unattractive to users. Even if a user does not immediately notice minor spacing problems, the human eye is very finicky about what text it considers acceptable. Poor spacing makes text harder to read because readers must strain their eyes to read the text. Properly spaced text makes readers—and their eyes—happier than poorly spaced text does.

Drawing Support for Raster Output

Table 15.3 summarizes the methods of the `Graphics` class that draw raster data. We define raster graphics as those functions that operate on an array of pixels. Two of the listed functions copy an icon (`DrawIcon`) or a bitmap (`DrawImage`) to a drawing surface. The other two methods fill a rectangular area with the color of an indicated brush. We discuss the details of creating and drawing with bitmaps later in this chapter.

Drawing Support for Vector Output

Table 15.4 summarizes the seven methods in the `Graphics` class that draw vector `Graphics` objects in the .NET Compact Framework. There

TABLE 15.2: `System.Drawing.Graphics` Methods for Text Drawing

Method	Comment
`DrawString`	Draws a single line of text using a specified font and text color.
`MeasureString`	Calculates the width and height of a specific character string using a specific font.

TABLE 15.3: `System.Drawing.Graphics` Methods for Raster Drawing

Method	Comment
`Clear`	Accepts a color value and uses that value to fill the entire surface of a window or the entire surface of a bitmap.
`DrawIcon`	Draws an icon at a specified location. An icon is a raster image created from two rectangular bitmap masks. The `DrawIcon` method draws an icon by applying one of the masks to the drawing surface using a Boolean AND operator, followed by the use of the XOR operator to apply the second mask to the drawing surface. The benefit of icons is that they allow portions of an otherwise rectangular image to display the screen behind the icon. The disadvantage of icons is that they are larger than comparable bitmaps and also slower to draw.
`DrawImage`	Draws a bitmap onto the display screen or draws a bitmap onto the surface of another bitmap.
`FillRegion`	Fills a region with the color specified in a brush. A region is defined as a set of one or more rectangles joined by Boolean operations.

TABLE 15.4: `System.Drawing.Graphics` Methods for Vector Drawing

Method	Comment
`DrawEllipse`	Draws the outline of an ellipse using a pen.
`DrawLine`	Draws a straight line using a pen.
`DrawPolygon`	Draws the outline of a polygon using a pen.
`DrawRectangle`	Draws the outline of a rectangle using a pen.
`FillEllipse`	Fills the interior of an ellipse using a brush.
`FillPolygon`	Fills the interior of a polygon using a brush.
`FillRectangle`	Fills the interior of a rectangle using a brush.

are substantially fewer supported vector methods than in the desktop .NET Framework. The vector methods whose names start with `Draw` draw lines. The vector methods whose names start with `Fill` fill areas.

Drawing on the Display Screen

The various `System.Drawing` classes in the .NET Compact Framework exist for two reasons. The first and most important reason is for output to the display screen. The second reason, which exists to support the first reason, is to enable drawing to bitmaps, which can later be displayed on the display screen.

Taken together, the various classes in the `System.Drawing` namespace support all three families of graphical output: text, raster, and vector. You can draw text onto the display screen using a variety of sizes and styles of fonts. You can draw with raster functions, including functions that draw icons, functions that draw bitmaps, and functions that fill regions[10] or the entire display screen. The third family of graphical functions, vector functions, supports the drawing of lines, polygons, rectangles, and ellipses on the display screen.

Accessing a `Graphics` Object

For a .NET Compact Framework program to draw on the display screen, it must have an instance of the `Graphics` class—meaning, of course, a `Graphics` object. A quick visit to the online documentation in the MSDN Library shows two interesting things about the `Graphics` class. First, this class provides no public constructors. Second, this class cannot be inherited by other classes. Thus you might wonder how to access a `Graphics` object.

Close study of the .NET Compact Framework classes reveals that there are three ways to access a `Graphics` object. Two are for drawing on a display screen, and one is for drawing on a bitmap. Table 15.5 summarizes three methods that are needed to gain access to a `Graphics` object. We

10. A region is a set of rectangles. Regions exist primarily to support clipping but can also be used to define an area into which one can draw.

TABLE 15.5: .NET Compact Framework Methods for Accessing a `Graphics` Object

Namespace	Class	Method	Comment
`System.Drawing`	`Graphics`	`FromImage`	Creates a `Graphics` object for drawing onto a bitmap. When done drawing, clean up the `Graphics` object by calling the `Dispose` method.
	`Graphics`	`Dispose`	Reclaims memory used by `Graphics` objects.
`System.Windows.Forms`	`Control`	`CreateGraphics`	Creates a `Graphics` object for drawing in the client area of a control. As indicated in Table 15.6, only three control classes support this method. When done drawing, clean up the `Graphics` object by calling the `Dispose` method.
	`Control`	`Paint` event handler	Obtains a `Graphics` object to handle a `Paint` event. As indicated in Table 15.6, only five control classes support this event. Do not call the `Dispose` method when done drawing.

include a fourth method in the table, `Dispose`, because you need to call that method to properly dispose of a `Graphics` object in some circumstances.

The display screen is a shared resource. A multitasking, multithreaded operating system like Windows CE needs to share the display screen and avoid conflicts between programs. For that reason, Windows CE uses the same mechanism used by Windows on the desktop: Drawing on a display screen is allowed only in a window (i.e., in a form or a control).

To draw on the display screen, a program draws in a control. You get access to a `Graphics` object for the display screen, then, through controls. Not just any control class can provide this access, however—only the control classes that derive from the `Control` class can.

One way to get a `Graphics` object for the display screen involves the `Paint` event. The `Paint` event plays a very important role in the design of the Windows CE user interface, a topic we discuss later in this chapter. Access to a `Graphics` object is provided to a `Paint` event handler method as a property of its `PaintEventArgs` parameter. Incidentally, when you get a `Paint` event, you are allowed to use the `Graphics` object while responding to the event. You are not allowed to hold onto a reference to the `Graphics` object because the .NET Compact Framework needs to recycle the contents of that `Graphics` object for other controls to use.[11]

A second way to get a `Graphics` object is by calling the `CreateGraphics` method, a method defined in the `Control` class (and therefore available to classes derived from the `Control` class). Using the `Graphics` object returned by this call, your program can draw inside a control's client area. Although the method name suggests that it is creating a `Graphics` object, this is not what happens. Instead, like the `Graphics` object that arrives with the `Paint` event, the `Graphics` object that is provided by the `CreateGraphics` method is loaned to you from a supply created and owned by the Windows CE window manager. Therefore, you are required to return this object when you are done by calling the `Graphics` object's `Dispose` method. Failure to make this call results in a program hanging.

11. Ultimately, the window manager reuses the device context contained within the `Graphics` object.

Calling the `Dispose` Method for a `Graphics` Object

There are two ways to get a `Graphics` object, but you need to call the `Dispose` method for only one of those ways. You must call the `Dispose` method for `Graphics` objects that are returned by the `CreateGraphics` method. But you do not call `Dispose` for `Graphics` objects that are provided as a parameter to the `Paint` event handler.

The third way to get a `Graphics` object is by calling the static `FromImage` method in the `Graphics` class. On the desktop, the `Image` class is an abstract class that serves as the base class for the `Bitmap` and `Metafile` classes. Because metafiles are not supported in the .NET Compact Framework, the `FromImage` method can return only a `Graphics` object for a bitmap. You can use the resulting `Graphics` object to draw onto a bitmap in the same way that the `Graphics` object described earlier is used to draw on a display screen. We are going to discuss drawing to bitmaps later in this chapter; for now, we explore the subject of drawing in controls.

As we discussed in Chapter 7, a main theme for .NET Compact Framework controls is that "inherited does not mean supported." Of the 28 available .NET Compact Framework control classes, only 5 support drawing. To help understand what types of drawing are supported, we start by identifying the specific controls that you can draw onto. We then cover the most important control event for drawing, the `Paint` event. We then discuss how non–`Paint` event drawing differs from `Paint` event handling.

Drawing in Controls

In the desktop .NET Framework, a program can draw onto any type of control (including onto forms). This feature is sometimes referred to as *owner-draw support*, a feature first seen in native-code programming for just a few of the Win32 API controls. The implementers of the .NET Framework for the desktop seem to think that this feature is something that every control should support. On the desktop, every control supports the owner-draw feature. In other words, you can get a `Graphics` object for every type

of control[12] and use that object to draw inside the client area of any control. Owner-draw support is widely available because it allows programmers to inherit from existing control classes and change the behavior and appearance of those classes. This support allows the creation of custom control classes from existing control classes.

Things are different in the .NET Compact Framework, for reasons that are directly attributable to the .NET Compact Framework design goals. As we discussed in detail in Chapter 7, the .NET Compact Framework itself was built to be as small as possible and also to allow .NET Compact Framework programs to run with reasonable performance. The result is a set of controls with the following qualities:

- .NET Compact Framework controls rely heavily on the built-in, Win32 API control classes.
- .NET Compact Framework controls do not support every PME inherited from the base `Control`[13] class.

The result is that only a few .NET Compact Framework controls provide owner-draw support. In particular, five control classes support the `Paint` event. Only three control classes support the `CreateGraphics` method. Table 15.6 summarizes the support for drawing in .NET Compact Framework control classes.

As suggested by the column headings in Table 15.6, there are two types of drawing: `Paint` event drawing and `CreateGraphics` method drawing. The clearest way to describe the difference is relative to events because of the unique role played by the `Paint` event and its associated `Paint` event handler method. From this perspective, the two types of drawing are better stated as `Paint` event drawing and drawing for other events. All five

12. All desktop control classes that we tested support the `CreateGraphics` method. However, a few desktop control classes do not support the overriding of the `Paint` event: the `ComboBox`, `HScrollbar`, `ListBox`, `ListView`, `ProgressBar`, `StatusBar`, `TabControl`, `TextBox`, `Toolbar`, `TrackBar`, `TreeView`, and `VScrollBar` classes.
13. Fully qualified name: `System.Windows.Forms.Control`.

TABLE 15.6: Support for Drawing in .NET Compact Framework Control Classes

Class	`Paint` Event	`CreateGraphics` Method
`Control`	Yes	Yes
`DataGrid`	Yes	Yes
`Form`	Yes	Yes
`Panel`	Yes	No
`PictureBox`	Yes	No

controls in Table 15.6 support `Paint` event drawing. We turn our attention now to the subject of the `Paint` event and its role in the Windows CE user interface.

Anywhere, Anytime Control Drawing

An early definition of .NET talked about "anywhere, anytime access to information." Arbitrary boundaries are annoying. It is odd, then, that you cannot draw onto your controls anywhere at any time. But wait—maybe you can?

If you are willing to step outside of the managed-code box, you can draw on any control at any time. The .NET Compact Framework team did a great job of giving us a small-footprint set of libraries with very good performance. That is why owner-draw support is so limited—not because of any belief on the part of the .NET Compact Framework team that you should not be allowed to draw inside controls.

Native-code drawing means using GDI calls, each of which requires you to have a handle to a *device context* (`hdc`). There are two types of device contexts: those used to draw inside windows and those that can draw anywhere on a display screen. To draw in a window, you first must get

the window handle (set focus to a control and then call the native `GetFocus` function). Call the native `GetDC` function to retrieve a device context handle, and call the `ReleaseDC` function when you are done.

A second method for accessing a device context is by using this call: `hdc = CreateDC(NULL, NULL, NULL, NULL)`. The device context that is returned provides access to the entire display screen, not just inside windows. Among its other uses, this type of device context is useful for taking screenshots of the display screen, which can be useful for creating documentation. When done with the device context, be sure to clean up after yourself by calling the `DeleteDC` function.

The `hdc` returned by either of these functions—`GetDC` or `CreateDC`—can be used as a parameter to any GDI drawing function. When done drawing, be sure to provide your own manual garbage collection. In other words, be sure to call the `ReleaseDC` or `DeleteDC` functions.

The `Paint` Event

To draw in a window—that is, in a form or in a control—you handle the `Paint`[14] event. This event is sent by the system to notify a window that the contents of the window need to be redrawn. In the parlance of Windows programmers, a window needs to be redrawn when some portion of its client area becomes *invalid*. To fix an invalid window, a control draws everything that it thinks ought to be displayed in the window.

Generating a `Paint` Event

The purpose of the `Paint` event is to centralize all the drawing for a window in one place. Before we look at more of the details of how to handle the `Paint` event, we need to discuss the circumstances under which a

14. This is what Win32 programmers know as a `WM_PAINT` message, which MFC programmers handle by overriding the `CWnd::OnPaint` method.

`Paint` event gets generated. A `Paint` event gets generated when the contents of a window become invalid. (We use the term *window* to mean a form or any control derived from the `Control` class.) But what causes a window to become invalid? There are several causes.

When a window is first created, its contents are invalid. When a form first appears, every control on the form is invalid. A `Paint` event is delivered to each control (which, in some cases, is handled by the native-code control that sits behind the managed-code control).

A window can also become invalid when it gets hidden. Actually, a hidden window is not invalid; it is just hidden. But when it gets uncovered, the window also becomes invalid. At that moment, a `Paint` event is generated by the system so that the window can repair itself.

A window can also become invalid when it gets scrolled. Every scroll operation causes three possible changes to the contents of a window. Some portion of the contents might disappear, which occurs when something scrolls off the screen. Nothing is required for that portion. Another portion might move because it has been scrolled up (or down, left, or right). Here again, nothing is required. The system moves that portion to the correct location. The third portion is the new content that is now visible to be viewed. This third portion must be drawn in response to a `Paint` event.

Finally, a `Paint` event is triggered when something in the logic of your program recognizes that the graphical display of a window does not match the program's internal state. Perhaps a new file was opened, or the user picked an option to zoom in (or out) of a view. Maybe the network went down (or came up), or the search for something ended.

To generate a `Paint` event for any window, a program calls one of the various versions of the `Invalidate` method for any `Control`-derived class. This method lets you request a `Paint` event for a portion of a window or for the entire window and optionally allows you to request that the background be erased prior to the `Paint` event.

This approach to graphical window drawing is not new to the .NET Compact Framework or even to the .NET environment. All GUI systems have a `Paint` event—from the first Apple Macintosh and the earliest versions of desktop Windows up to the current GUI systems shipping today. A window holds some data and displays a view of that data.

In one sense, drawing is simple: A window draws on itself using the data that it holds. And what happens if the data changes? In that case, a window must declare its contents to be invalid, which causes a `Paint` event to be generated. A control requests a `Paint` event by calling the `Invalidate` method. Two basic problems can be observed with the `Paint` event:

1. Failing to request `Paint` events (which causes cold windows with stale contents)
2. Requesting `Paint` events too often (which causes hot window flickers that annoy users)

These are different problems, but both involve calling the `Invalidate` method the wrong number of times. The first problem arises from not invalidating a window enough. The second problem arises from invalidating the window too often. A happy medium is needed: invalidating a window the right number of times and at just the right times.

To draw in response to a `Paint` event, a program adds a `Paint` event handler to a control. You can add a `Paint` event handler to any `Control`-derived class. But the handler is only going to get called for the five control classes listed in Table 15.6. This is just another example of the "inherited does not mean supported" behavior of .NET Compact Framework controls.

Here is an empty `Paint` event handler.

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // draw
}
```

The second parameter to the `Paint` event handler is an instance of `PaintEventArgs`.[15] A property of this class is a `Graphics` object, which provides the connection that we need to draw in the form. There is more to be said about the `Graphics` object, but first let us look at the case of drawing for events besides the `Paint` event.

Non–`Paint` Event Drawing

A window that contains any graphical output must handle the `Paint` event. Often, the only drawing that a window requires is the drawing for the `Paint` event. This is especially true if the contents of the window are somewhat static. For example, `Label` controls are often used to display text that does not change. For a `Label` control, drawing for the `Paint` event is all that is required. However, windows whose contents must change quickly might need to draw in response to events other than the `Paint` event. A program that displays some type of animation, for example, might draw in response to a `Timer` event. A program that echoes user input might draw in response to keyboard or mouse events.

Figure 15.1 shows the `DrawRectangles` program, a sample program we presented in Chapter 6. This program draws rectangles in the program's main form, using a pair of (x,y) coordinates. One coordinate pair is collected for the `MouseDown` event, and a second coordinate pair is collected for the `MouseUp` event. As the user moves the mouse (or a stylus on a Pocket PC), the program draws a stretchable rubber rectangle as the mouse/stylus is moved from the `MouseDown` point to the `MouseUp` point. The program accumulates rectangles as the user draws them.

The `DrawRectangles` program uses both `Paint` and non–`Paint` event drawing. In response to the `Paint` event, the program draws each of the accumulated rectangles. In response to the `MouseMove` event, the stretchable rectangle is drawn to allow the user to preview the result before committing to a specific location.

15. Fully qualified name: `System.Windows.Forms.PaintEventArgs`.

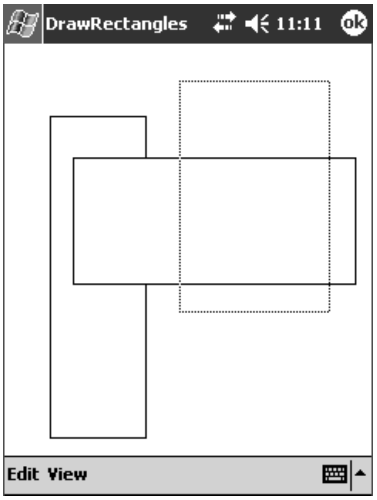

FIGURE 15.1: A stretchable rubber rectangle created in the DrawRectangles program

The basic template for the code used in non–`Paint` event drawing appears here.

```
Graphics g = CreateGraphics();
// Draw
g.Dispose();
```

This follows a programming pattern familiar to some as the *Windows sandwich*.[16] The top and bottom lines of code make up the two pieces of bread—these are always the same. The filling in between the two slices of bread consists of the drawing, which is accomplished with the drawing methods from the `Graphics` class.

Raster Graphics

We define raster graphics as those functions that operate on an array of pixels. The simplest raster operation is to fill a rectangle with a single color.

16. This is what Eric Maffei of *MSDN Magazine* used to refer to as the *Windows Hoagie*.

On a display screen, this is one of the most common operations. If you study any window on any display screen, you are likely to see that the background is a single color, often white or sometimes gray. You can use three methods in the `Graphics` class to fill a rectangular area:

- `Clear`: fills a window with a specified color
- `FillRectangle`: fills a specified rectangle using a brush
- `FillRegion`: fills a specified region using a brush

The `Clear` method accepts a single parameter, a structure of type `Color`.[17] The other two methods accept a `Brush`[18] object as the parameter that identifies the color to use to fill the area. Before we can fill an area with any of these functions, then, we need to know how to define colors and how to create brushes.

Specifying Colors

The most basic type of drawing attribute is color, yet few drawing methods accept colors directly as parameters. Most drawing methods require other drawing attributes that have a built-in color. For example, for filling areas, the color to use for filling the area is the color that is part of a brush. When drawing lines, the line color is the color that is part of a pen. Brushes are also used to specify text color. So even though color parameters are not directly used as parameters to methods, they are indirectly specified through a pen or brush.

There are three ways to specify a color in a .NET Compact Framework program:

- With a system color
- With a named color
- With an RGB value

17. Fully qualified name: `System.Drawing.Color`.
18. Fully qualified name: `System.Drawing.Brush`.

System colors are a set of colors used to draw the elements of the user interface. The use of system colors helps create consistency between different programs. For example, a given system might be set up with black text on a white background, or it could be set up with the opposite, white text on a black background. System colors are made available as properties of the `SystemColors`[19] class. Available system colors are listed in Table 15.7 in the System Colors subsection.

Named colors provide access to colors that use human-readable names like `Red`, `Green`, `Blue`, `White`, and `Black`. There are also a large number of colors with less common names like `SeaShell` and `PeachPuff`. Whether or not you like all the names of the colors you encounter, they provide a way to specify colors that is easy to remember. Color names are made available as static properties of the `Color` structure.

When you specify a color using an RGB value, you specify an amount of red, an amount of green, and an amount of blue. Each is defined with a byte, meaning that values can range from `0` to `255`. It is sometimes helpful to remember that RGB is a video-oriented color scheme often used for display screens and televisions. When the energy for all three colors is `0`, the color you see is black; when all the energy for all three colors is at 100% (`255`), the resulting color is white.

System Colors

System colors let you connect your program's graphical output to current system settings. This allows a program to blend in with the current system configuration. On some platforms, users can change system colors from the system control panel (such as on desktop version of Microsoft Windows). Other platforms, like the Pocket PC, do not provide the user with an easy way to modify system color settings. A custom embedded smart device could easily be created with a unique system color scheme—say, to match corporate logo colors or to meet unique environmental requirements such as usage in low-light conditions or in sunlight. For all of these cases, the safest approach to selecting text colors involves using system colors.

19. Fully qualified name: `System.Drawing.SystemColors`.

System colors are available as read-only properties in the `SystemColors` class, the contents of which are summarized in Table 15.7. If you study this table, you may notice that several entries have the word `Text` in the name—such as `ControlText` and `WindowText`. There are, after all, many uses of text in the user interface. When specifying the color for drawing text, these system colors provide your best choice.

TABLE 15.7: System Colors in the .NET Compact Framework

Color	Description
`ActiveBorder`	Border color of a window when the window is active
`ActiveCaption`	Color of the background in the caption when a window is active
`ActiveCaptionText`	Color of the text in the caption when a window is active
`AppWorkspace`	Color of the unused area in an MDI[a] application
`Control`	Background color for a three-dimensional control
`ControlDark`	Color of the middle of the shadow for a three-dimensional control
`ControlDarkDark`	Color of the darkest shadow for a three-dimensional control
`ControlLight`	Color of the lightest element in a three-dimensional control
`ControlLightLight`	Color of the lightest edge for a three-dimensional control
`ControlText`	Color for drawing text in controls
`Desktop`	Color of the desktop background
`GrayText`	Color for drawing grayed text (e.g., for disabled controls)
`Highlight`	Background color of highlighted areas for menus, `ListBox` controls, and `TextBox` controls
`HighlightText`	Text color for highlighted text
`HotTrack`	Color of hot-tracked items

a. MDI is not supported on Windows CE or in the .NET Compact Framework.

Color	Description
InactiveBorder	Border color of a top-level window when the window is inactive
InactiveCaption	Color of the background in the caption when a window is inactive
InactiveCaptionText	Color of the text in the caption when a window is inactive
Info	Background color of a tool tip
InfoText	Text color of a tool tip
Menu	Menu background color
MenuText	Menu text color
ScrollBar	Background color of a scroll bar
Window	Background color of a window
WindowFrame	Color of a window border
WindowText	Color of text in a window

In some cases, there is a pair of system color names: one with and one without the word `Text` in the name (e.g., `Control` and `ControlText`, `Window` and `WindowText`). One color in the pair defines a system color for text, and the other defines the color of the background. For example, when drawing in a form or dialog box, use the `Window` color for the background and `WindowText` for the text color. When you create a custom control, use the `Control` color for the control's background area and `ControlText` for the color of text drawn in a control. In the following code, the background is filled with the default window background color.

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.Clear(SystemColors.Window);
}
```

Named Colors

The `System.Drawing.Color` class defines 142 named colors as read-only properties. The names include old favorites like `Red`, `Green`, `Blue`, `Cyan`, `Magenta`, `Yellow`, `Brown`, and `Black`. It also includes some new colors like `AliceBlue`, `AntiqueWhite`, `Aqua`, and `Aquamarine`. With names like `Chocolate`, `Honeydew`, and `PapayaWhip`, you may get hungry just picking a color. The named colors appear in Table 15.8.

The following code draws in the background of a window with the color `PapayaWhip`.

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.Clear(Color.PapayaWhip);
}
```

Colors from RGB Values

The third approach that the .NET Compact Framework supports for specifying colors is to specify the three components—red, green, and blue—that make up a color. These three components are packed together into a 32-bit integer with one byte for each. The range for each component is from `0` to `255` (`FF` in hexadecimal). Table 15.9 summarizes color triplet values for common colors.

To create a color from an RGB triplet, use the `Color.FromArgb` method. There are two overloaded versions for this method. We find the following one easier to use.

```
public static Color FromArgb(
   int red,
   int green,
   int blue);
```

When you read the online documentation for this method, you see a reference to a fourth element in a color, the *alpha value*. The .NET Compact Framework does not support this, so you can safely ignore it. (In the desktop .NET Framework, the alpha value defines the transparency of a color, where a value of `0` is entirely transparent and `255` is entirely opaque. In a

TABLE 15.8: Named Colors in the .NET Compact Framework

AliceBlue	DarkGray	Gold	LightSkyBlue	Navy	SandyBrown
AntiqueWhite	DarkGreen	Goldenrod	LightSlateGray	OldLace	SeaGreen
Aqua	DarkKhaki	Gray	LightSteelBlue	Olive	SeaShell
Aquamarine	DarkMagenta	Green	LightYellow	OliveDrab	Sienna
Azure	DarkOliveGreen	GreenYellow	Lime	Orange	Silver
Beige	DarkOrange	Honeydew	LimeGreen	OrangeRed	SkyBlue
Bisque	DarkOrchid	HotPink	Linen	Orchid	SlateBlue
Black	DarkRed	IndianRed	Magenta	PaleGoldenrod	SlateGray
BlanchedAlmond	DarkSalmon	Indigo	Maroon	PaleGreen	Snow
Blue	DarkSeaGreen	Ivory	MediumAquamarine	PaleTurquoise	SpringGreen
BlueViolet	DarkSlateBlue	Khaki	MediumBlue	PaleVioletRed	SteelBlue
Brown	DarkSlateGray	Lavender	MediumOrchid	PapayaWhip	Tan
BurlyWood	DarkTurquoise	LavenderBlush	MediumPurple	PeachPuff	Teal
CadetBlue	DarkViolet	LawnGreen	MediumSeaGreen	Peru	Thistle
Chartreuse	DeepPink	LemonChiffon	MediumSlateBlue	Pink	Tomato
Chocolate	DeepSkyBlue	LightBlue	MediumSpringGreen	Plum	Transparent
Coral	DimGray	LightCoral	MediumTurquoise	PowderBlue	Turquoise
CornflowerBlue	DodgerBlue	LightCyan	MediumVioletRed	Purple	Violet
Cornsilk	Firebrick	LightGoldenrodYellow	MidnightBlue	Red	Wheat
Crimson	FloralWhite	LightGray	MintCream	RosyBrown	White
Cyan	ForestGreen	LightGreen	MistyRose	RoyalBlue	WhiteSmoke
DarkBlue	Fuchsia	LightPink	Moccasin	SaddleBrown	Yellow
DarkCyan	Gainsboro	LightSalmon	NavajoWhite	Salmon	YellowGreen
DarkGoldenrod	GhostWhite	LightSeaGreen			

TABLE 15.9: Color Triplets for Common Colors

Color Name	RGB Triplet (Decimal)	RGB Triplet (Hexadecimal)
Black	`(0, 0, 0)`	`(0, 0, 0)`
White	`(255, 255, 255)`	`(0xFF, 0xFF, 0xFF)`
Red	`(255, 0, 0)`	`(0xFF, 0, 0)`
Green	`(0, 255, 0)`	`(0, 0xFF, 0)`
Blue	`(0, 0, 255)`	`(0, 0, 0xFF)`
Cyan	`(0, 255, 255)`	`(0, 0xFF, 0xFF)`
Magenta	`(255, 0, 255)`	`(0xFF, 0, 0xFF)`
Yellow	`(255, 255, 0)`	`(0xFF, 0xFF, 0)`
Dark Gray	`(68, 68, 68)`	`(0x44, 0x44, 0x44)`
Medium Gray	`(128, 128, 128)`	`(0x80, 0x80, 0x80)`
Light Gray	`(204, 204, 204)`	`(0xCC, 0xCC, 0xCC)`

Knowing Black from White

We give programmers in our training classes the following tip to help remember the correct RGB for black `(0, 0, 0)` and white `(255, 255, 255)`. The RGB color encoding is a light-based scheme, which in a computer CRT is often used to correlate the power to apply to the electron guns in the monitor. Turn the power off, which causes the power to go to zero, and you see black. When the power to the red, green, and blue is turned up all the way, you get white.

In Table 15.9, notice the different shades of gray. By studying the color triplets, you can observe what makes the color gray: equal parts of red, green, and blue.

.NET Compact Framework program, all colors have an alpha value of `255`, which means that all colors are 100% opaque.

The following code draws the window background using a light gray color.

```
private void FormMain_Paint(
object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.Clear(Color.FromArgb(204,204,204));
}
```

Creating Brushes

A brush specifies the color and pattern to use for area-filling methods, such as `FillRectangle`. The .NET Compact Framework does not support patterns in brushes, however, so a brush just specifies the color when filling areas. Brushes also specify the color to use when drawing text. The second parameter to the `DrawString` method, for example, is a brush.

The desktop .NET Framework supports five different kinds of brushes, including solid brushes, bitmap brushes, and hatch brushes. Windows CE supports solid brushes and bitmap brushes but not hatch brushes. And in the .NET Compact Framework, things are even simpler: only solid brushes are supported, by the `SolidBrush`[20] class. This class has a single constructor, which takes a single parameter—`Color`. The `SolidBrush` constructor is defined as follows.

```
public SolidBrush(
   Color color);
```

With one constructor, it is natural to assume that there is one way to create a solid brush. But because there are three ways to define a color, there are three ways to create a brush:

- Using the system colors
- Using a named color
- Using an RGB value

20. Fully qualified name: `System.Drawing.SolidBrush`.

The following subsections discuss each of these briefly.

Creating Brushes with System Colors

The following code creates a brush from a system color. This brush is suitable for drawing text within a program's main form or in a dialog box.

```
Brush brText = new SolidBrush(SystemColors.WindowText);
```

The resulting brush provides the same color used by the operating system to draw text. You are not required to select this color, but in doing so you help ensure that your application fits into the color scheme established by the user.

There might be reasons to design your own color scheme. For example, when dealing with financial figures you might display positive numbers in black and display negative numbers in red. Or perhaps when displaying certain types of documents you could highlight keywords in different colors, in the same way that Visual Studio .NET highlights language keywords in blue. To handle these situations, you need to specify the brush color with one of the two other color-defining schemes: using either named colors or RGB colors.

Creating Brushes with Named Colors

Here are examples of creating brushes using named colors.

```
Brush brRed = new SolidBrush(Color.Red);
Brush brPaleGreen = new SolidBrush(Color.PaleGreen);
Brush brLightBlue = new SolidBrush(Color.LightBlue);
```

You might wonder where these color names come from. Some—like `Red`—are, of course, names for common colors. But when you read through the list of names, you see colors like `AliceBlue`, `GhostWhite`, and `WhiteSmoke`. The colors are sometimes called *HTML Color Names* because the more exotic names were first supported as color names in HTML by various browsers. Officially, however, HTML 4.0 includes only 16 color names, not the 140+ names defined in the `Color` structure.

Creating Brushes with RGB Values

To create a brush using an RGB value, call the `FromArgb` method in the `Color` class and pass the return value to the `SolidBrush` constructor. This method accepts three integer parameters, one each for red, green, and blue. Here is how to create three brushes from RGB triplets.

```
Brush brRed = new SolidBrush(Color.FromArgb(255, 0, 0));
Brush brGreen = new SolidBrush(Color.FromArgb(0, 255, 0));
Brush brBlue = new SolidBrush(Color.FromArgb(0, 0, 255));
```

Creating Bitmaps

A bitmap is a two-dimensional array of pixels with a fixed height and a fixed width. Bitmaps have many uses. One is to hold scanned images, such as a company logo. Photographs are stored as bitmaps, commonly in the highly compressed format of JPEG[21] files. Bitmaps can be used to create interesting effects on a display screen, such as smooth scrolling and seamless animation.

Bitmaps are often used to store complex images that a program can easily draw in one or more locations by making a single method call. As useful as this approach can be, it is important to always remember that bitmaps require a lot of room—both in memory and in the file system. If you plan to include any bitmaps with your program, give some thought to the format of those bitmaps. We address this issue later in this chapter.

Bitmaps are sometimes referred to as *off-screen-bitmaps* because of the important role bitmaps have historically played in supporting display screen graphics. The Bitmaps on the Desktop sidebar discusses how bitmaps are used on desktop versions of Windows to support various user interface objects. That same support does not exist in Windows CE because of memory constraints. But bitmaps are still available to Windows CE programs for all of their other uses.

21. *JPEG* stands for Joint Photographic Experts Group, the name of the original committee that created the standard. For details on this compression standard, visit http://www.jpeg.org.

Bitmaps on the Desktop

On desktop versions of Windows, bitmaps support the quick appearance and disappearance of menus, dialog boxes, and various other user interface elements. For example, before a menu appears on the screen, a snapshot is taken of the area to be covered by the menu. When the menu disappears, the bitmap is used to redraw the affected part of the screen. This technique helps make the elements of the user interface appear and disappear very quickly. This technique is not employed in Windows CE because of the tight memory restrictions of mobile and embedded systems. But your program could use bitmaps in other ways to support the display of your program's user interface.

In our programming classes, we observe that programmers often get confused when first starting to work with bitmaps. The confusion seems to come from not grasping that bitmaps are inherently off-screen. Or it may arise from an understanding that display screens are supported by memory-mapped video devices and that the memory occupied by a bitmap must somehow be related to the memory used by the display adapter. After creating a bitmap and drawing into a bitmap, some programmers expect that bitmaps are going to appear somewhere on the screen. That does not happen, however, because bitmaps appear on a display screen only when your program explicitly causes them to appear.

Bitmaps: Drawing Surface or Drawing Object?

Bitmaps play two roles in every graphic library built for Microsoft Windows: (1) as drawing surfaces and (2) as drawing objects used to draw onto other surfaces. This is another reason why bitmaps can at first seem confusing for some programmers.

A bitmap is a drawing surface like other drawing surfaces. We say this because a program can obtain a `Graphics` object for a bitmap and then use the methods of that object to draw onto the surface of the bitmap. All of the

drawing methods in the `Graphics` object are supported for bitmaps, including text, raster, and vector drawing methods.

The second role played by bitmaps is that of a drawing object. Like other drawing objects, such as pens, brushes, and fonts, a bitmap holds a pattern that can be applied to a drawing surface. Each drawing object has its particular uses, and each produces a different effect, as determined by the various drawing methods. The .NET Compact Framework supports four overloads for the bitmap drawing method, which is named `DrawImage`.

An example might clarify what we mean by each of these two roles. Using a `Graphics` object, a program can draw onto a bitmap by calling drawing methods. One such drawing method is `DrawImage`, which draws a bitmap onto a drawing surface. A program can call the `DrawImage` method to draw one bitmap (the drawing object) onto the surface of another bitmap (the drawing surface).

To push the example one step further, a bitmap can be both the drawing surface and also the drawing object. You could do this by using the `DrawImage` method to draw onto a bitmap while using the bitmap itself as the image source. This may sound like a snake eating its own tail, a seemingly impossible operation. It is possible, however, because it involves copying a rectangular array of pixels from one part of a bitmap to another part. The work required for this type of bitmap handling is well understood and has been part of Windows display drivers for more than a decade. The bitmap drawing code in Windows CE can easily—and correctly—handle cases where, for example, source and destination rectangles overlap. This describes what happens, for example, when a user picks up and moves a window.

The `Bitmap` Class

The .NET Compact Framework supports in-memory bitmaps with the `Bitmap`[22] class. This class is derived from the `Image` class, which is a common base class for the `Bitmap` class and the `Metafile` class. As we mentioned earlier in this chapter, metafiles are not supported in the .NET

22. Fully qualified name: `System.Drawing.Bitmap`.

Compact Framework. But because the .NET Compact Framework maintains consistency with the desktop .NET Framework, our bitmap drawing method is called `DrawImage` (instead of, for example, `DrawBitmap`). On the desktop, where metafiles are supported, the `DrawImage` method draws both bitmaps and metafiles.

The dimensions of a bitmap are available through two properties of the `Bitmap` object: `Height` and `Width`. The dimensions are also available through the `Size` property, which provides a convenient package for height and width. These are read-only properties because an image cannot change size once it has been created.

On the desktop, the `Bitmap` class supports 12 constructors, while in the .NET Compact Framework there are only 4 constructors. You can create a bitmap these ways:

- By opening an image file
- By reading an image from a stream
- By starting from an existing bitmap
- By specifying the width and height for an empty bitmap

Table 15.10 maps the constructors to six common sources you might use to create a bitmap.

Creating an Empty Bitmap

One way to create a bitmap is to specify the desired dimensions of the bitmap to the following constructor in the `Bitmap` class.

```
public Bitmap(
   int width,
   int height);
```

This constructor creates a bitmap in program memory with the specified size. This is the quickest and easiest way to create a bitmap, but the empty (i.e., all-black) image means that you must draw into the bitmap before displaying its contents. You might call this a *scratch space* or *double-buffer bitmap* because it provides an off-screen drawing surface for doodling, just like scratch paper. The term *double-buffer* refers to a technique of creating

TABLE 15.10: Sources for Bitmaps and Associated `Bitmap` Class Constructors

Source	Constructor Parameters	Comments
An external image file	`(String)`	Provides the path to the bitmap in the file system
A portion of a file	`(Stream)`	Uses the `FileStream` class to open the file and move the seek position to the first byte of the bitmap
Data in memory	`(Stream)`	Uses the `MemoryStream` class to assemble the bitmap bits as a byte array
A resource	`(Stream)`	Reads bitmap data from a managed resource created as an untyped manifest resource
An existing bitmap	`(Image)`	Copies an existing bitmap
An empty bitmap	`(int,int)`	Specifies the width and height of the empty bitmap

smooth graphic effects by doing complex drawing off-screen and sending the resulting output to the display screen with a single, fast drawing operation. Let's use a bitmap created with this constructor.

After creating the bitmap itself, a program typically obtains a `Graphics` object for the bitmap. As we mentioned earlier in this chapter, we need a `Graphics` object for any type of drawing. We obtain a `Graphics` object for the bitmap by calling the `FromImage` method of the `Bitmap` class. Before drawing anything else in the bitmap, it makes sense to first erase the bitmap's background.

We need to think about cleanup. This is a subject that can often be ignored in managed code, but not when working with resource-intensive objects like bitmaps. So, when done working with a bitmap, your program must use the `Dispose` method to clean up two objects: the bitmap itself and the `Graphics` object. The code in Listing 15.1 shows the whole life cycle of our created bitmap: The code creates a bitmap, erases the bitmap's background, draws the bitmap to the display screen, and then cleans up the two objects that were created.

LISTING 15.1: Dynamic Bitmap Creation

```
private void
CreateAndDraw(int x, int y)
{
   // Create a bitmap and a Graphics object for the bitmap.
   Bitmap bmpNew = new Bitmap(100,100);
   Graphics gbmp = Graphics.FromImage(bmpNew);

   // Clear the bitmap background.
   gbmp.Clear(Color.LightGray);

   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();

   // Copy the bitmap to the window at (x,y) location.
   g.DrawImage(bmpNew, x, y);

   // Clean up when we are done.
   g.Dispose();
   gbmp.Dispose();
   bmpNew.Dispose();
}
```

Creating a Bitmap from an External File

Another way to create a bitmap is by specifying the path to an image file. This is accomplished with a constructor that accepts a single parameter, a string with the path to the candidate file. This second `Bitmap` class constructor is defined as follows.

```
public Bitmap(
   string filename);
```

This method has two important requirements. One is that there must be enough memory to accommodate the bitmap. If there is not, the call fails. A second requirement is that the specified file must have an image in a format that the constructor understands. We have been able to create bitmaps from the following file types:

- Bitmap files (`.bmp`) with 1, 4, 8, or 24 bits per pixel
- JPEG (`.jpg`) files

- GIF (`.gif`) files
- PNG (`.png`) files

Among the unsupported graphic file formats are TIFF (`.tif`) files.

This constructor throws an exception if the file name provided is not a recognized format or if it encounters other problems when attempting to open the file or create the bitmap. For that reason, it makes sense to wrap this constructor in a `try`…`catch` block. Listing 15.2 provides an example of calling this constructor, with a file name provided by the user in a File Open dialog box.

LISTING 15.2: Creating a Bitmap with a File Name

```
try
{
   bmpNew = new Bitmap(strFileName);
}
catch
{
   MessageBox.Show("Cannot create bitmap from " +
      "File: " + strFileName);
}
```

Creating a Bitmap from a Resource

When a program needs a bitmap for its normal operation, it makes sense to package the bitmap as a resource. Resources are read-only data objects that are bound into a program's executable file[23] at program build time. The benefit of binding a bitmap to an executable file is that it is always available and cannot be accidentally deleted by a user.

Resources have been a part of Windows programming from the very first version of Windows. In a native-mode program, resources are used for bitmaps and icons and also to hold the definitions of dialog boxes and menus. In managed-code programs, resources are still used for bitmaps and icons, although some program elements—including dialog boxes and

23. Resources can be bound into any executable module, meaning any program (`.exe`) or library (`.dll`) file.

menus—are not defined in a resource but instead are defined using code in the `InitializeComponent` method by the Designer.

Where Do Resources Live?

Because memory is scarce on a Windows CE–powered device, it helps to know when and how memory gets used. When a resource gets added to a module, the resource occupies space in the module's file but uses no program memory until the resource is explicitly opened and used. This is true for both native resources and managed resources.

While resources are used in both native code and managed code, native resources can be used only from native mode code, and managed resources can be used only from managed code. The only exception is the program icon for a managed-code program, which is defined as a native icon. In managed code, there are two types of resources: *typed resources* and *untyped resources*.

Typed Resources. We like to use typed resources to hold literal strings, which aid in the localization of programs. To access typed resources, a program creates an instance of a `ResourceManager`[24] class and then makes calls to methods like `GetObject` and `GetString`. We provided an example of using typed resources for literal strings in Chapter 3, in the sample project named `StringResources`.

Typed resources are defined using XML in files that have an extension of `.resx`. In a typed resource, an XML attribute provides type information, as shown in this example.

```
<data name="dlgFileOpen.Location" type="System.Drawing.Point,
 System.CF.Drawing, Version=7.0.5000.0, Culture=neutral,
 PublicKeyToken=b03f5f7f11d50a3a">
     <value>125, 17
     </value>
</data>
```

24. Fully qualified name: `System.Resources.ResourceManager`.

The Designer makes extensive use of typed resources. For each form created in the Designer, there is an associated file used to store a variety of details about the form. The Visual Studio .NET Solution Explorer does not normally display resource files, but you can make them appear by clicking the Show All Files icon.

For example, bitmaps in the image collection of an `ImageList` control on a form are stored as typed resources in the typed resource file of the form that contains the control. The bitmaps themselves are serialized into the XML and are not stored in their original, binary form. While a programmer could convert bitmap files into XML resources, we prefer to avoid this extra step and use untyped resources when we add bitmap resources to a project.

Untyped Resources. Untyped resources are also known as *manifest resources* because they are made available through an assembly's manifest (or table of contents). As the name implies, an untyped resource contains no type information and is made available as a raw stream of bytes. It does have a name, however, created by combining the default project namespace with the file name that contained the original resource data. You must know this name because you use the name to retrieve the resource. If you have trouble figuring out the resource name, the `ildasm.exe` utility can help. Open the program file and then click on the manifest. Listing 15.3 shows three bitmap resource names in a fragment from the manifest for the `ShowBitmap` sample program presented later in this chapter.

LISTING 15.3: Three Bitmap Resource Names from the `ShowBitmap` Manifest

```
.mresource public ShowBitmap.SPADE.BMP
{
}
.mresource public ShowBitmap.CUP.BMP
{
}
.mresource public ShowBitmap.HEART.BMP
{
}
```

Visual Studio .NET creates an untyped resource from an external file when you add the file to a project and assign a build action of Embedded Resource. The default build action for bitmap files, Content, allows the bitmap file to be downloaded with a program, but as a separate file and not as an embedded resource. Figure 15.2 shows the Visual Studio .NET settings to turn the file `CUP.BMP` into an embedded bitmap resource. The name of the resource is `ShowBitmap.CUP.BMP`, which we need to know to access the resource from our code.

You can access an embedded resource by calling a method in the `Assembly` class named `GetManifestResourceStream`.[25] As suggested by the method name, the return value is a `Stream` object; more precisely, you are provided a `MemoryStream`[26] object. You can use all of the elements associated with a `Stream`-derived class (including the ability to query the resource length, which is the same as the resource input file) to seek a location in the stream and to read bytes (the `CanSeek` and `CanRead` properties are both set to `true`). In keeping with the read-only nature of Windows resources, you cannot write to a resource stream[27] (`CanWrite` returns `false`).

The code fragment in Listing 15.4 shows two methods from the `ShowBitmap` sample program. These methods are helper routines to handle the

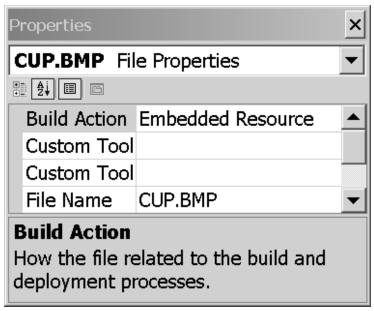

FIGURE 15.2: The settings to turn `CUP.BMP` into an embedded bitmap resource

25. Fully qualified name: `System.Reflection.Assembly.GetManifestResourceStream`.
26. Fully qualified name: `System.IO.MemoryStream`.
27. In contrast to an Apple Macintosh resource fork, which supports both read and write operations.

What Can Go into an Untyped Resource?

This chapter provides an example of putting a bitmap into an untyped resource. But this is not the only type of resource you can create. You can put any custom data into untyped resources, which can then be used to access the data at runtime. When you request an untyped resource, you are provided with a `Stream` object that you can use as you wish. You might, for example, read a resource into an array of bytes and then parse those bytes in whatever way your application needs. Such resources can be any read-only data that your program needs: tax tables, sports scores, or—as we show in the `ShowBitmap` sample program—a set of bitmaps.

A benefit of using custom resources is that we have access to data we need at runtime. But when we are not using that data, it does not occupy scarce program memory. This makes custom resources a useful tool in our toolkit for building memory-wise programs.

initialization and cleanup of resource-based bitmaps. The `LoadBitmapResource` method creates a bitmap from a resource; the `DisposeBitmap` method provides the cleanup.

LISTING 15.4: Creating Bitmaps from Untyped Manifest Resources

```
private Bitmap LoadBitmapResource(string strName)
{
   Assembly assembly = Assembly.GetExecutingAssembly();
   string strRes = "ShowBitmap." + strName;
   Stream stream = assembly.GetManifestResourceStream(strRes);
   Bitmap bmp = null;
   try
   {
      bmp = new Bitmap(stream);
   }
   catch { }
   stream.Close();

   return bmp;
}
```

continues

continued

```
private void DisposeBitmap(ref Bitmap bmp)
{
   if (bmp != null)
   {
      bmp.Dispose();
   }
   bmp = null;
}
```

The `LoadBitmapResource` method creates a bitmap by opening a resource stream and uses data read from that stream to create a bitmap. This method gets a reference to the program's assembly by calling a static method in the `Assembly` class named `GetExecutingAssembly`. After creating a bitmap, the stream can be closed. Once a bitmap has been created, it is self-contained and needs no external data. That is why we can close the stream once the `Bitmap` object has been created.

The `DisposeBitmap` method deletes the bitmap to free up its associated memory. It does this by calling the `Dispose` method for a `Bitmap` object. There are only a few situations in which it is mandatory to call the `Dispose` method.[28] Sometimes, however, it is still a good idea—even if it is not, strictly speaking, required. Bitmaps can be large, so we suggest you consider explicitly deleting bitmaps, as we have done in our sample. Among the factors to consider are the size of your bitmaps and the number of bitmaps. We suggest that you explicitly delete bitmaps when you have either many small bitmaps or a few large bitmaps.

We call our two methods using code like the following.

```
private void
mitemResourceCup_Click(object sender, EventArgs e)
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("CUP.BMP");
   Invalidate();
}
```

28. The only situation requiring a call to the `Dispose` method is to release a `Graphics` object obtained in a control by calling the `CreateGraphics` method.

After cleaning up the old bitmap, we create a new bitmap and request a `Paint` event by calling the `Invalidate` method. Next, we discuss image file size, and how to save memory by changing the format you use for your images.

Image File Sizes

Bitmaps can occupy a lot of memory, which can create problems in a memory-scarce environment like Windows CE. When placing bitmaps in resources, we recommend that you test different formats and use the smallest one. To provide a starting point, we conducted some tests with three 100×100 pixel images stored in different formats. Table 15.11 summarizes our results, which provide the size in bytes for each image file.

Four formats are uncompressed and three are compressed. The first four entries in the table are for DIB files. This well-known format is thoroughly documented in the MSDN Library and is the format that Visual Studio .NET provides for creating bitmap images. Notice that the size of these images is the same for a given number of bits per pixel. This reflects the fact that DIB files are uncompressed.

TABLE 15.11: Size Comparison for Three 100 x 100 Images in Various Image File Formats

Format	Bits per Pixel	Size of Single-Color Image (Bytes)	Size of Multicolor Image with Regular Data (Bytes)	Size of Multi-color Image with Irregular Data (Bytes)
Monochrome DIB	1	1,662	1,662	1,662
16-color DIB	4	5,318	5,318	5,318
256-color DIB	8	11,078	11,078	11,078
True-color DIB	24	30,054	30,054	30,054
GIF	8	964	3,102	7,493
PNG	8	999	616	5,973
JPEG	24	823	3,642	5,024

The last three formats are the compressed formats: GIF, PNG, and JPEG. To make sense of these formats, we must discuss the contents of the three images. The single-color image was a solid black rectangle. Each of the three compressed formats easily beat any of the uncompressed formats for the single-color image. The reason is that compressed formats look for a pattern and use that information to store details of the pattern. A single color is a pretty easy pattern to recognize and compress.

The second column, the multicolor image with regular data, shows the results for an image created with a solid background and vertical stripes. We used vertical stripes in an attempt to thwart the compression because run-length encoding of horizontal scan lines is an obvious type of compression. We were surprised (and pleased) to find that PNG compression was able to see through the fog we so carefully created—it created the smallest image in the table.

The third column, the multicolor image with irregular data, shows the sizes for images created with very random data. For this test, we copied text (.NET Compact Framework source code) into an image file. (We never want our work to be called "random," but we wanted an irregular image to push the envelope for the three compression formats.) The result was more like a photograph than any of the other images, which is why JPEG—the compression scheme created for photographs—was able to provide the best compression. It provided the smallest file size with the least loss of information (the monochrome image was smaller, but the image was lost).

To summarize, the two compression schemes that created the smallest image files were PNG (for regular data) and JPEG (for irregular data). One problem is that Visual Studio .NET does not support either of these formats. But Microsoft Paint (`mspaint.exe`) supports both, so we recommend that you make sure your images have been compressed as much as possible prior to embedding your images as resources.

Drawing Bitmaps

The `Graphics` class supports four overloaded versions of the bitmap drawing method, `DrawImage`. These alternatives support the following types of bitmap drawing:

- Drawing the entire bitmap at the original image size
- Drawing part of a bitmap at the original image size
- Drawing part of a bitmap with a change to the image size
- Drawing part of a bitmap with a change to the image size and with transparency

We discuss these four methods in the sections that follow.

Drawing the Entire Bitmap at the Original Image Size

The simplest version of the `DrawImage` method copies an entire bitmap onto a device surface with no change in the image size, as shown here.

```
public void DrawImage(
   Image image,
   int x,
   int y);
```

Listing 15.5 shows an example of calling this method in a `Paint` event handler.

LISTING 15.5: Drawing an Entire Bitmap at the Original Size

```
private void
FormMain_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   int x = 10;
   int y = 10;

   g.DrawImage(bmpDraw, x, y);
}
```

Drawing Part of a Bitmap at the Original Image Size

While we sometimes want to draw an entire bitmap, there are also times when we only want to see a portion of a bitmap. The second version of the `DrawImage` method provides the support we need to do just that, as shown on the next page.

```
public void DrawImage(
   Image image,
   int x,
   int y,
   Rectangle srcRect,
   GraphicsUnit srcUnit);
```

This version of the `DrawImage` method has five parameters, while the earlier one has only three. One of the extra parameters is useful, and the second is not so useful. The fourth parameter, `srcRect`, is the useful one, which identifies the rectangular area in the source bitmap that we wish to copy to the destination surface.

The fifth parameter, `srcUnit`, can be set to only one valid value in the .NET Compact Framework: `GraphicsUnit.Pixel`. On the desktop, the presence of this parameter gives the caller the freedom to select a convenient unit of measure for the source rectangle (e.g., inches or millimeters). But the .NET Compact Framework supports only pixel drawing units, which is why this parameter is not so useful in the context of a .NET Compact Framework program. The `srcUnit` parameter is present because of the high level of compatibility between the desktop .NET Framework and the .NET Compact Framework. As such, it represents a small price to pay for the convenience of allowing smart-device code to have binary compatibility with the desktop runtime.

Drawing Part of a Bitmap with a Change to the Image Size

The third overloaded version of the `DrawImage` method allows a portion of a bitmap to be selected for drawing, and that portion can be stretched (or shrunk) to match a specified size on the destination surface. Of course, nothing requires the image to change size: If the width and height of the destination rectangle is the same as the width and height of the source rectangle, no size change occurs. This version of the `DrawImage` method is defined as shown here.

```
public void DrawImage(
   Image image,
   Rectangle destRect,
   Rectangle srcRect,
   GraphicsUnit srcUnit);
```

Drawing Part of a Bitmap with a Change to the Image Size and with Transparency

The final version of the `DrawImage` method adds a new feature to the drawing of bitmaps. It enables transparency while drawing a bitmap. In some ways, this feature breaks our definition of raster graphics. You might recall that we refer to raster graphics as those operations that operate on arrays of pixels. Implicit in this definition is that all operations are rectangular.

The ability to draw a raster operation and touch only a nonrectangular set of pixels on a drawing surface is, therefore, something of a heresy (like having nonrectangular windows on a display screen or a late-night coding session without ordering large quantities of unhealthy food). We hope that readers can accept this change with little loss of sleep. We certainly are happy to break the shackles that have previously limited almost all raster graphics to the boring world of rectangular arrays of pixels. This amazing new feature is available through the following version of the `DrawImage` method.

```
public void DrawImage(
   Image image,
   Rectangle destRect,
   int srcX,
   int srcY,
   int srcWidth,
   int srcHeight,
   GraphicsUnit srcUnit,
   ImageAttributes imageAttr);
```

With its eight parameters, this version of the `DrawImage` method is the most complicated one that the .NET Compact Framework supports. Perhaps it is appropriate that this version matches the other versions in capabilities: It can draw an entire bitmap at its original size, draw a portion of a bitmap at its original size, and draw a portion of a bitmap at a different size.

What makes this version different is the final parameter, a reference to an `ImageAttributes` object. On the desktop, this class supports a variety of color adjustments that can be applied when drawing a bitmap onto a surface. The .NET Compact Framework version is much simpler, with what amounts to a single property: a color key. The color key defines the range of colors that represent transparent portions of an image. In other

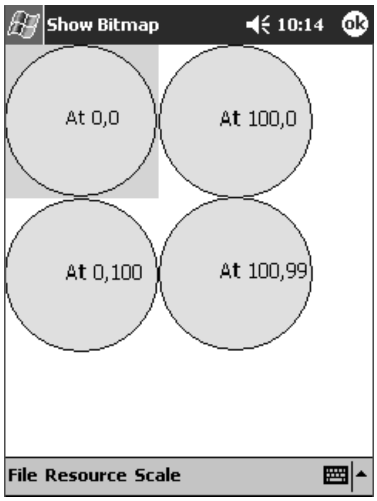

FIGURE 15.3: Four calls to the `DrawImage` method, three with transparency enabled

words, any color that matches the color key is a color that is *not* copied by the call to the `DrawImage` method. The color key settings are controlled through two methods: `SetColorKey` defines the transparency range, and `ClearColorKey` disables the transparency range.

Figure 15.3 shows an example of transparency at work. A 100 × 100 bitmap is first drawn without transparency at the window origin. That same bitmap is then drawn three times, using the version of the `DrawImage` method that supports transparency. The color key is set to light gray, which corresponds to the color outside the ellipse (the interior of the ellipse is set to yellow). Listing 15.6 shows the code, a handler for a `MouseDown` event, which we used to create the example.

LISTING 15.6: Event Handler That Draws a Bitmap with Transparency

```
bool bFirstTime = true;

private void
FormMain_MouseDown(object sender, MouseEventArgs e)
{
   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();
```

```
    // Create a bitmap and a Graphics object for the bitmap.
    Bitmap bmpNew = new Bitmap(100,100);
    Graphics gbmp = Graphics.FromImage(bmpNew);

    // Clear the bitmap background.
    gbmp.Clear(Color.LightGray);

    // Create some drawing objects.
    Pen penBlack = new Pen(Color.Black);
    Brush brBlack = new SolidBrush(Color.Black);
    Brush brYellow = new SolidBrush(Color.Yellow);

    // Draw onto the bitmap.
    gbmp.FillEllipse(brYellow, 0, 0, 98, 98);
    gbmp.DrawEllipse(penBlack, 0, 0, 98, 98);
    gbmp.DrawString("At " + e.X.ToString() + "," + e.Y.ToString(),
       Font, brBlack, 40, 40);

    // Copy the bitmap to the window at the MouseDown location.
    if (bFirstTime)
    {
       // Copy without transparency.
       g.DrawImage(bmpNew, e.X, e.Y);
       bFirstTime = false;
    }
    else
    {
       // Copy the bitmap using transparency.
       Rectangle rectDest = new Rectangle(e.X, e.Y, 100, 100);
       ImageAttributes imgatt = new ImageAttributes();
       imgatt.SetColorKey(Color.LightGray, Color.LightGray);
       g.DrawImage(bmpNew, rectDest, 0, 0, 99, 99,
          GraphicsUnit.Pixel, imgatt);
    }

    // Clean up when we are done.
    g.Dispose();
    gbmp.Dispose();
    bmpNew.Dispose();
}
```

A Sample Program: `ShowBitmap`

Our bitmap drawing sample program shows several features of bitmaps that we have been discussing. This program can open files and create a bitmap. Several formats are supported, including the standard Windows DIB (`.bmp`)

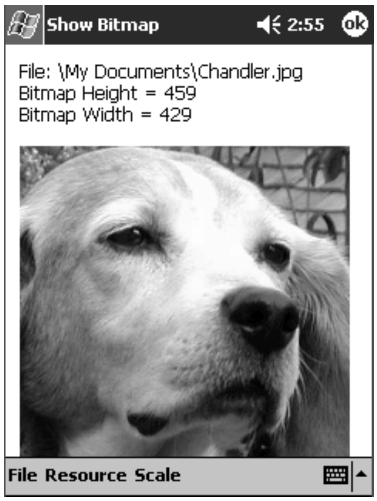

FIGURE 15.4: ShowBitmap displaying a JPEG file scaled to 50%

files and also a few compressed image file formats such as GIF (`.gif`) files, JPEG (`.jpg`) files, and PNG (`.png`) files. Figure 15.4 shows the `ShowBitmap` program with a JPEG image of Chandler (the office beagle at The Paul Yao Company). This image is drawn scaled to 50%, an effect made possible by selecting the appropriate version of the `DrawImage` method.

Our sample program contains a set of bitmap files that are bound to the program files as embedded resources (see Listing 15.7). As with all types of resources, the resource data does not get loaded into memory until we explicitly load the resource. In this program, we load the resource when the user selects an item on the program's resource menu. Figure 15.5 shows the bitmap resource that was read from a resource identified as `ShowBitmap.CUP.BMP`, drawn at 400% of its original size.

LISTING 15.7: Source Code for `ShowBitmap.cs`

```
using System.Reflection; // Needed for Assembly
using System.IO;         // Needed for Stream
using System.Drawing.Imaging;  // Needed for ImageAttributes
// ...
```

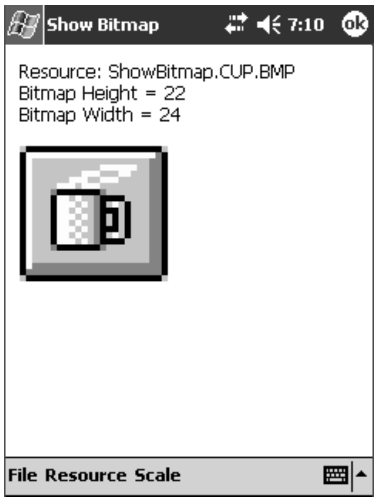

FIGURE 15.5: `ShowBitmap` displaying a bitmap from a resource

```
private Bitmap bmpDraw;
bool bFirstTime = true;
bool bResource = false;
string strResName;

// Draw a bitmap using transparency where the MouseDown
// event is received.
private void
FormMain_MouseDown(object sender, MouseEventArgs e)
{
#if false
   CreateAndDraw(e.X, e.Y);
#endif
   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();

   // Create a bitmap and a Graphics object for the bitmap.
   Bitmap bmpNew = new Bitmap(100,100);
   Graphics gbmp = Graphics.FromImage(bmpNew);

   // Clear the bitmap background.
   gbmp.Clear(Color.LightGray);

   // Create some drawing objects.
   Pen penBlack = new Pen(Color.Black);
```

continues

continued

```
   Brush brBlack = new SolidBrush(Color.Black);
   Brush brYellow = new SolidBrush(Color.Yellow);

   // Draw onto the bitmap.
   gbmp.FillEllipse(brYellow, 0, 0, 98, 98);
   gbmp.DrawEllipse(penBlack, 0, 0, 98, 98);
   gbmp.DrawString("At " + e.X.ToString() + "," + e.Y.ToString(),
      Font, brBlack, 40, 40);

   // Copy the bitmap to the window at the MouseDown location.
   if (bFirstTime)
   {
      // Copy without transparency.
      g.DrawImage(bmpNew, e.X, e.Y);
      bFirstTime = false;
   }
   else
   {
      // Copy the bitmap using transparency.
      Rectangle rectDest = new Rectangle(e.X, e.Y, 100, 100);
      ImageAttributes imgatt = new ImageAttributes();
      imgatt.SetColorKey(Color.LightGray, Color.LightGray);
      g.DrawImage(bmpNew, rectDest, 0, 0, 99, 99,
         GraphicsUnit.Pixel, imgatt);
   }

   // Clean up when we are done.
   g.Dispose();
   gbmp.Dispose();
   bmpNew.Dispose();
}

private void FormMain_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   float sinX = 10.0F;
   float sinY = 10.0F;
   SizeF szfText = g.MeasureString("X", Font);
   float cyLine = szfText.Height;

   Brush brText = new SolidBrush(SystemColors.WindowText);
   if (bmpDraw != null)
   {
      if (bResource)
      {
         g.DrawString("Resource: " + strResName,
            Font, brText, sinX, sinY);
      }
      else
```

```
    {
        g.DrawString("File: " + dlgFileOpen.FileName,
            Font, brText, sinX, sinY);
    }
    sinY += cyLine;

    g.DrawString("Bitmap Height = " + bmpDraw.Height,
        Font, brText, sinX, sinY);
    sinY += cyLine;

    g.DrawString("Bitmap Width = " + bmpDraw.Width,
        Font, brText, sinX, sinY);
    sinY += cyLine;
    sinY += cyLine;

    if (mitemScale100.Checked)
    {
        g.DrawImage(bmpDraw, (int)sinX, (int)sinY);
    }
    else
    {
        Rectangle rectSrc = new Rectangle(0, 0,
            bmpDraw.Width, bmpDraw.Height);
        int xScaled = 0;
        int yScaled = 0;
        if (mitemScale50.Checked)
        {
            xScaled = bmpDraw.Width / 2;
            yScaled = bmpDraw.Height / 2;
        }
        else if (mitemScale200.Checked)
        {
            xScaled = bmpDraw.Width * 2;
            yScaled = bmpDraw.Height * 2;
        }
        else if (mitemScale400.Checked)
        {
            xScaled = bmpDraw.Width * 4;
            yScaled = bmpDraw.Height * 4;
        }

        Rectangle rectDest = new Rectangle((int)sinX,
            (int)sinY, xScaled, yScaled);
        g.DrawImage(bmpDraw, rectDest, rectSrc,
            GraphicsUnit.Pixel);
    }
}
else
```

continues

continued

```
   {
      g.DrawString("File: None", Font, brText, sinX, sinY);
   }
}

private void
mitemFileOpen_Click(object sender, EventArgs e)
{
   dlgFileOpen.Filter = "Bitmap (*.bmp)|*.bmp|" +
                        "Picture (*.jpg)|*.jpg|" +
                        "PNG Files (*.png)|*.png|" +
                        "TIF Files (*.tif)|*.tif|" +
                        "GIF Files (*.gif)|*.gif |" +
                        "All Files (*.*)|*.*";
   if (dlgFileOpen.ShowDialog() == DialogResult.OK)
   {
      Bitmap bmpNew = null;
      try
      {
         bmpNew = new Bitmap(dlgFileOpen.FileName);
         bResource = false;
      }
      catch
      {
         MessageBox.Show("Cannot create bitmap from " +
            "File: " + dlgFileOpen.FileName);
         return;
      }

      DisposeBitmap (ref bmpDraw);
      bmpDraw = bmpNew;
      Invalidate();
   }
}

private void
mitemScale_Click(object sender, EventArgs e)
{
   // Clear the checkmark on related items.
   mitemScale50.Checked = false;
   mitemScale100.Checked = false;
   mitemScale200.Checked = false;
   mitemScale400.Checked = false;

   // Set the checkmark on selected menu item.
   ((MenuItem)sender).Checked = true;
```

```
    // Request paint to redraw bitmap.
    Invalidate();
}


private void
mitemResourceCup_Click(object sender, EventArgs e)
{
    DisposeBitmap(ref bmpDraw);
    bmpDraw = LoadBitmapResource("CUP.BMP");
    Invalidate();
}

private void
mitemResourceBell_Click(object sender, EventArgs e)
{
    DisposeBitmap(ref bmpDraw);
    bmpDraw = LoadBitmapResource("BELL.BMP");
    Invalidate();
}

private void
mitemResourceSpade_Click(object sender, EventArgs e)
{
    DisposeBitmap(ref bmpDraw);
    bmpDraw = LoadBitmapResource("SPADE.BMP");
    Invalidate();
}

private void
mitemResourceHeart_Click(object sender, EventArgs e)
{
    DisposeBitmap(ref bmpDraw);
    bmpDraw = LoadBitmapResource("HEART.BMP");
    Invalidate();
}

private void
mitemResourceDiamond_Click(object sender, EventArgs e)
{
    DisposeBitmap(ref bmpDraw);
    bmpDraw = LoadBitmapResource("DIAMOND.BMP");
    Invalidate();
}

private void
mitemResourceClub_Click(object sender, EventArgs e)
```

continues

continued

```
{
   DisposeBitmap(ref bmpDraw);
   bmpDraw = LoadBitmapResource("CLUB.BMP");
   Invalidate();
}

private Bitmap LoadBitmapResource(string strName)
{
   Assembly assembly = Assembly.GetExecutingAssembly();
   string strRes = "ShowBitmap." + strName;
   Stream stream = assembly.GetManifestResourceStream(strRes);
   Bitmap bmp = null;
   try
   {
      bmp = new Bitmap(stream);
      strResName = strRes;
      bResource = true;
   }
   catch { }
   stream.Close();

   return bmp;
}

private void DisposeBitmap(ref Bitmap bmp)
{
   if (bmp != null)
   {
      bmp.Dispose();
   }
   bmp = null;
}

// Simplest possible bitmap: Create a bitmap, clear the
// bitmap background, draw the bitmap to the display screen.
private void
CreateAndDraw(int x, int y)
{
   // Create a bitmap and a Graphics object for the bitmap.
   Bitmap bmpNew = new Bitmap(100,100);
   Graphics gbmp = Graphics.FromImage(bmpNew);

   // Clear the bitmap background.
   gbmp.Clear(Color.LightGray);

   // Get a Graphics object for the form.
   Graphics g = CreateGraphics();
```

```
    // Copy the bitmap to the window at (x,y) location.
    g.DrawImage(bmpNew, x, y);

    // Clean up when we are done.
    g.Dispose();
    gbmp.Dispose();
    bmpNew.Dispose();
}
```

Vector Graphics

The available vector drawing methods in the .NET Compact Framework are summarized in Table 15.12 (which appeared earlier in this chapter as Table 15.4 and is repeated here for convenience). As indicated in the table, some shapes are drawn with a pen, a drawing object used for lines. The .NET Compact Framework supports only pens that are 1 pixel wide (unless a programmer drills through to the native GDI drawing support). Other shapes in the table are drawn with a brush. We discussed the three methods for creating brushes earlier in this chapter. We cover the creation of pens in this discussion of vector graphics.

TABLE 15.12: `System.Drawing.Graphics` Methods for Vector Drawing

Method	Comment
`DrawEllipse`	Draws the outline of an ellipse using a pen.
`DrawLine`	Draws a straight line using a pen.
`DrawPolygon`	Draws the outline of a polygon using a pen.
`DrawRectangle`	Draws the outline of a rectangle using a pen.
`FillEllipse`	Fills the interior of an ellipse using a brush.
`FillPolygon`	Fills the interior of a polygon using a brush.
`FillRectangle`	Fills the interior of a rectangle using a brush.

The vector methods with names that start with `Draw` are those that use a pen to draw a line or a set of connected lines. The call to the `DrawRectangle` method, for example, draws the outline of a rectangle without touching the area inside the line. If you pass a blue pen to the `DrawRectangle` method, the result is the outline of a rectangle drawn with a blue line. The .NET Compact Framework supports four line-drawing methods.

Vector methods whose names start with `Fill`, on the other hand, use a brush to fill in the area bounded by the lines. For example, if you pass a red brush to the `FillRectangle` method, the result is a solid red rectangle. There are three such methods in the .NET Compact Framework for drawing ellipses, polygons, and rectangles.

The `Draw` and `Fill` methods complement each other. You could, for example, pass a red brush to the `FillRectangle` method and pass a blue pen to the `DrawRectangle` method using the same coordinates that you used to draw the red, filled rectangle. The result would be a two-colored rectangle, with a blue border and a red interior. This type of two-colored figure is natively available in the Windows API. Yet it seems apparent that few programs need to draw two-colored vector figures. That is, no doubt, a factor that contributed to the design of vector drawing in the .NET Framework and the .NET Compact Framework.

If a programmer is willing to do a bit of work, almost all vector drawing can be accomplished by calling two of these methods: `DrawLine` and `FillPolygon`. Each of the supported method names is of the form <verb><shape>. In the `DrawLine` method, for example, the verb is `Draw` and the shape is `Line`.

Creating Pens

Pens draw lines. The desktop supports a very sophisticated model for pens, including support for scalable geometric pens and nonscalable cosmetic pens. Pens on the desktop support features that allow you to fine-tune how an end of a line appears (rounded or squared) and even how the "elbow" joints are drawn. Pens can be wide or narrow, and even nonsolid pen colors are supported.

Wake up! In the .NET Compact Framework, pens are always 1 pixel wide. Pens provide a quick and simple way to define the color used to

draw a line. From the seventeen properties supported for pens on the desktop, one has survived to the .NET Compact Framework: `Color`. And so it should come as no surprise that the one constructor for the `Pen`[29] class has a single parameter, a color as shown here.

```
public Pen(
   Color color);
```

There are three ways to define a pen in a .NET Compact Framework program because there are three ways to specify a color:

1. With a system color
2. With a named color
3. With an RGB value

Earlier in this chapter, we described some of the details about the three ways to pick a color. We showed that each of the color-specifying approaches could be used to create a brush. Now the time has come to show the same thing for pens.

The following code fragment creates three pens. One pen is created using a system color; another pen is created using a named color; and finally, the third pen is created with an RGB value.

```
// Pen from a system color
Pen penCtrl = new Pen(SystemColors.ControlDark);

// Pen from a named color
Pen penRed  = new Pen(Color.Red);

// Pen from an RGB value
Pen penBlue = new Pen(Color.FromArgb(0, 0, 255));
```

A Game: `JaspersDots`

While writing this book, we watched Paul's son, Jasper, playing a paper-and-pencil game with one of his friends. They were having so much fun

29. Fully qualified name: `System.Drawing.Pen`.

that we decided to write a .NET Compact Framework version. The game was Dots, which may be familiar to some readers. In this two-person game, players take turns connecting dots that have been drawn in a grid. A player is awarded a point for drawing the last line that creates a box. We named our version of the game `JaspersDots`, in honor of Paul's son. The playing board for this game is drawn entirely with the following vector graphic methods:

- `FillEllipse`
- `FillRectangle`
- `DrawLine`
- `DrawRectangle`

This program provides extensive use of various `Graphics` objects including colors, pens, and brushes.

Figure 15.6 shows the New Game dialog box. Each player enters a name and picks a color to use for claimed squares. The default board size is 8×8, which can be overridden in the New Game dialog box (the maximum board size is 11×9).

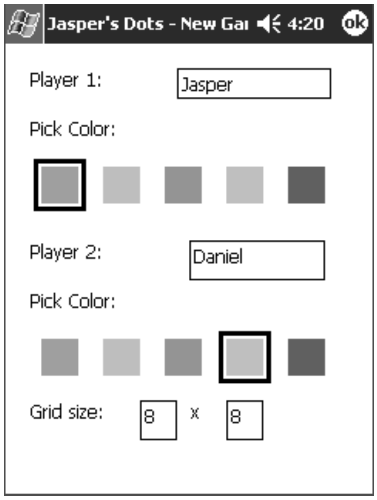

FIGURE 15.6: New Game dialog box for the `JaspersDots` program

The New Game dialog box is a simple dialog box drawn with regular controls, with one small enhancement: This dialog handles a `Paint` event, which draws a selection rectangle around each player's currently selected color. The set of available colors is drawn with `Panel` controls, five for each player. Listing 15.8 shows the source code for the event handlers for responding to the `Click` event for each set of `Panel` controls and to the `Paint` event for the New Game dialog box.

LISTING 15.8: `Paint` and `Click` Event Handlers for the New Game Dialog Box

```
private void
Panel1_Click(object sender, EventArgs e)
{
   if (sender == (object)panel1)
      iColor1 = 0;
   else if (sender == (object)panel2)
      iColor1 = 1;
   else if (sender == (object)panel3)
      iColor1 = 2;
   else if (sender == (object)panel4)
      iColor1 = 3;
   else if (sender == (object)panel5)
      iColor1 = 4;

   // Redraw the window.
   Invalidate();
}

private void
Panel2_Click(object sender, EventArgs e)
{
   if (sender == (object)panelA)
      iColor2 = 0;
   else if (sender == (object)panelB)
      iColor2 = 1;
   else if (sender == (object)panelC)
      iColor2 = 2;
   else if (sender == (object)panelD)
      iColor2 = 3;
   else if (sender == (object)panelE)
      iColor2 = 4;

   // Redraw the window.
   Invalidate();
}
```

continues

continued

```
private void
GameNewDialog_Paint(object sender, PaintEventArgs e)
{
   Panel panel = panel1;

   //
   // Player 1
   //
   // What is the current player 1 panel?
   switch(iColor1)
   {
      case 0:
         panel = panel1;
         break;
      case 1:
         panel = panel2;
         break;
      case 2:
         panel = panel3;
         break;
      case 3:
         panel = panel4;
         break;
      case 4:
         panel = panel5;
         break;
   }
   clr1 = panel.BackColor;

   // Draw a rectangle around the color selected by player 1.
   Pen penBlack = new Pen(Color.Black);
   Rectangle rc = new
      Rectangle(panel.Left - 3,
      panel.Top - 3,
      panel.Width + 5,
      panel.Height + 5);
   e.Graphics.DrawRectangle(penBlack, rc);
   rc.Inflate(1, 1);
   e.Graphics.DrawRectangle(penBlack, rc);
   rc.Inflate(1, 1);
   e.Graphics.DrawRectangle(penBlack, rc);

   //
   // Player 2
   //
   // What is the current player 2 panel?
```

```
    switch(iColor2)
    {
       case 0:
          panel = panelA;
          break;
       case 1:
          panel = panelB;
          break;
       case 2:
          panel = panelC;
          break;
       case 3:
          panel = panelD;
          break;
       case 4:
          panel = panelE;
          break;
    }
    clr2 = panel.BackColor;

    // Draw a rectangle around the color selected by player 2.
    rc = new Rectangle(panel.Left - 3,
       panel.Top - 3,
       panel.Width + 5,
       panel.Height + 5);
    e.Graphics.DrawRectangle(penBlack, rc);
    rc.Inflate(1, 1);
    e.Graphics.DrawRectangle(penBlack, rc);
    rc.Inflate(1, 1);
    e.Graphics.DrawRectangle(penBlack, rc);
}
```

There is a bug in Visual Studio .NET that affects C# programmers. The bug is that supported events for certain controls do not appear in the Designer. You can, however, add an event handler manually. Inside the Visual Studio code editor, you type the control name, the event name, and the `+=` operator, and IntelliSense helps by providing the rest.

In our `JaspersDots` game, we found that the Designer did not support the `Click` event for `Panel` controls. To create `Click` event handlers for the `Panel` controls in the New Game dialog box, we manually typed in event handler names, which were completed for us by IntelliSense. The resulting code appears in Listing 15.9.

LISTING 15.9: Adding Event Handlers Manually

```
// Set up the Click handler for player 1 panels.
// Note: The Designer does not support this
// so we have to do it manually.
panel1.Click += new EventHandler(this.Panel1_Click);
panel2.Click += new System.EventHandler(this.Panel1_Click);
panel3.Click += new System.EventHandler(this.Panel1_Click);
panel4.Click += new System.EventHandler(this.Panel1_Click);
panel5.Click += new System.EventHandler(this.Panel1_Click);

// Set up the Click handler for player 2 panels.
// Note: The Designer does not support this
// so we have to do it manually.
panelA.Click += new EventHandler(this.Panel2_Click);
panelB.Click += new System.EventHandler(this.Panel2_Click);
panelC.Click += new System.EventHandler(this.Panel2_Click);
panelD.Click += new System.EventHandler(this.Panel2_Click);
panelE.Click += new System.EventHandler(this.Panel2_Click);
```

Figure 15.7 shows an example of the `JaspersDots` game in play. Each dot is drawn with a call to the `FillEllipse` method that is drawn in a bounding rectangle that is 4 pixels by 4 pixels. Players draw lines by clicking in the area between dots, and when a hit is detected a line is drawn by

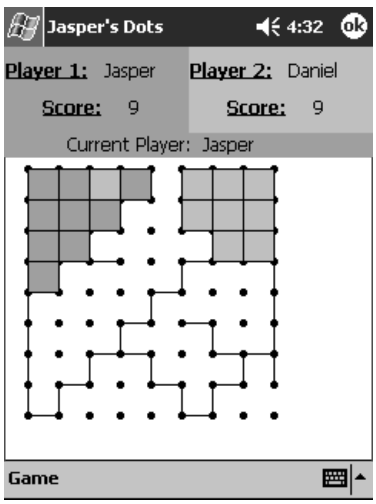

FIGURE 15.7: `JaspersDots` with a game under way

calling the `DrawLine` method. A player's claimed boxes are drawn with calls to the `FillRectangle` method.

The `JaspersDots` program uses a custom control for the game window, our `DotControl` class. Listing 15.10 shows the source code to the `DotControl` class.

LISTING 15.10: The `DotControl` Class

```
public class DotControl : System.Windows.Forms.Control
{
   private FormMain formParent;
   private Brush m_brPlayer1;
   private Brush m_brPlayer2;
   private Squares sq;

   public DotControl(FormMain form)
   {
      formParent = form;

      formParent.Controls.Add(this);
      this.Paint += new
         PaintEventHandler(this.DotControl_Paint);
      this.MouseDown += new
         MouseEventHandler(this.DotControl_MouseDown);
      this.Left = 0;
      this.Top = 64;
      this.Width = 240;
      this.Height = 240;

      sq = new Squares(this);
   }

   public bool SetGridSize(int cxWidth, int cyHeight)
   {
      return sq.SetGridSize(cxWidth, cyHeight);
   }

   public bool SetPlayerColors(Color clr1, Color clr2)
   {
      m_brPlayer1 = new SolidBrush(clr1);
      m_brPlayer2 = new SolidBrush(clr2);

      return sq.SetPlayerBrushes(m_brPlayer1, m_brPlayer2);
   }


   private void
   DotControl_MouseDown(object sender, MouseEventArgs e)
```

continues

continued

```
    {
        // Check result.
        int iResult = sq.HitTest(e.X, e.Y,
            formParent.CurrentPlayer);

        // Click on the available line, no score.
        if(iResult == 1)
        {
            formParent.NextPlayer();
        }

        // Click on the available line, score.
        if (iResult == 2)
        {
            int iScore1 = sq.GetScore(1);
            formParent.DisplayScore(1, iScore1);
            int iScore2 = sq.GetScore(2);
            formParent.DisplayScore(2, iScore2);

            int count = sq.Height * sq.Width;
            if (iScore1 + iScore2 == count)
            {
                string strResult = null;

                if (iScore1 > iScore2)
                    strResult = "Player 1 wins! ";
                else if (iScore1 < iScore2)
                    strResult = "Player 2 wins! ";
                else
                    strResult = "Tie Game! ";

                MessageBox.Show(strResult, "JaspersDots");
            }
        }
    }

    private void
    DotControl_Paint(object sender, PaintEventArgs e)
    {
        // Fill squares which players now own.
        sq.FillSquares(e.Graphics);

        // Draw lines which players have selected.
        sq.DrawLines(e.Graphics);

        // Draw dots in grid.
        sq.DrawDots(e.Graphics);
    }

} // class
```

The `DotControl` class handles two events: `MouseDown` and `Paint`. Most of the work for these events is done by a helper class named `Squares`. The source code for the `Squares` class appears in Listing 15.11.

LISTING 15.11: The `Squares` Class

```
public class Squares
{
   public int Width
   {
      get   { return cxWidth; }
   }
   public int Height
   {
      get   { return cyHeight; }
   }

   private int cxLeft = 15;
   private int cyTop  = 15;
   private int cxWidth;
   private int cyHeight;
   const int cxLine = 20;
   const int cyLine = 20;
   const int cxyDelta = 5;
   private Square [,] m_asq;

   private Control m_ctrlParent;
   private Brush m_brPlayer1;
   private Brush m_brPlayer2;
   private Brush m_brBackground = new _
      SolidBrush(SystemColors.Window);
   private Brush hbrBlack = new SolidBrush(Color.Black);
   private Point ptTest = new Point(0,0);
   Rectangle rc = new Rectangle(0, 0, 0, 0);
   private Size  szDot = new Size(4,4);

   Pen penLine = new Pen(Color.Black);

   public Squares(Control ctrlParent)
   {
      m_ctrlParent = ctrlParent;
   } // Squares()

   public bool SetGridSize(
      int cxNewWidth,     // Width of array.
      int cyNewHeight     // Height of array.
      )
```

continues

continued

```
{
   // Temporary scratch space
   Rectangle rcTemp = new Rectangle(0,0,0,0);
   Point     ptTemp = new Point(0,0);
   Size      szTemp = new Size(0,0);

   // Set up an array to track squares.
   cxWidth = cxNewWidth;
   cyHeight = cyNewHeight;
   m_asq = new Square[cxWidth, cyHeight];
   if (m_asq == null)
      return false;

   int x, y;
   for (x = 0; x < cxWidth; x++)
   {
      for (y = 0; y < cyHeight; y++)
      {
         m_asq[x,y].iOwner = 0; // No owner.
         int xLeft = cxLeft + x * cxLine;
         int yTop = cyTop + y * cyLine;
         int xRight = cxLeft + (x+1) * cxLine;
         int yBottom = cyTop + (y+1) * cyLine;
         int cxTopBottom = cxLine - (2 * cxyDelta);
         int cyTopBottom = cxyDelta * 2;
         int cxLeftRight = cxyDelta * 2;
         int cyLeftRight = cxLine - (2 * cxyDelta);

         // Main rectangle
         ptTemp.X = xLeft + 1;
         ptTemp.Y = yTop + 1;
         szTemp.Width = xRight - xLeft - 1;
         szTemp.Height = yBottom - yTop - 1;
         rcTemp.Location = ptTemp;
         rcTemp.Size = szTemp;
         m_asq[x,y].rcMain = rcTemp;

         // Top hit rectangle
         m_asq[x,y].rcTop =
            new Rectangle(xLeft + cxyDelta,
            yTop - cxyDelta,
            cxTopBottom,
            cyTopBottom);
         m_asq[x,y].bTop = false;

         // Right hit rectangle
         m_asq[x,y].rcRight =
            new Rectangle(xRight - cxyDelta,
            yTop + cxyDelta,
```

```
            cxLeftRight,
            cyLeftRight);
         m_asq[x,y].bRight = false;

         // Bottom hit rectangle
         m_asq[x,y].rcBottom =
            new Rectangle(xLeft + cxyDelta,
            yBottom - cxyDelta,
            cxTopBottom,
            cyTopBottom);
         m_asq[x,y].bBottom = false;

         // Left hit rectangle
         m_asq[x,y].rcLeft =
            new Rectangle(xLeft - cxyDelta,
            yTop + cxyDelta,
            cxLeftRight,
            cyLeftRight);
         m_asq[x,y].bLeft = false;

      } // for y
   } // for x

   return true;
}

public bool
SetPlayerBrushes(
   Brush br1,      // Brush color for player 1
   Brush br2       // Brush color for player 2
   )
{
   m_brPlayer1 = br1;
   m_brPlayer2 = br2;

   return true;
}

//-------------------------------------------------------
public void
FillOneSquare(Graphics g, int x, int y)
{
   Brush brCurrent = m_brBackground;
   if (m_asq[x,y].iOwner == 1)
      brCurrent = m_brPlayer1;
   else if (m_asq[x,y].iOwner == 2)
      brCurrent = m_brPlayer2;
   g.FillRectangle(brCurrent, m_asq[x,y].rcMain);
}
```

continues

continued

```
// FillSquares -- Fill owned squares with a player's color.
//
public void
FillSquares(Graphics g)
{
   int x, y;
   for (x = 0; x < cxWidth; x++)
   {
      for (y = 0; y < cyHeight; y++)
      {
         if (m_asq[x,y].iOwner != 0)
         {
            FillOneSquare(g, x, y);
         }
      }
   }
} // method: FillSquares

//
// DrawOneLineSet
//
public void DrawOneLineSet(Graphics g, int x, int y)
{
   int xLeft = cxLeft + x * cxLine;
   int yTop = cyTop + y * cyLine;
   int xRight = cxLeft + (x+1) * cxLine;
   int yBottom = cyTop + (y+1) * cyLine;

   if (m_asq[x,y].bTop)
      g.DrawLine(penLine, xLeft, yTop, xRight, yTop);
   if (m_asq[x,y].bRight)
      g.DrawLine(penLine, xRight, yTop, xRight, yBottom);
   if (m_asq[x,y].bBottom)
      g.DrawLine(penLine, xRight, yBottom, xLeft, yBottom);
   if (m_asq[x,y].bLeft)
      g.DrawLine(penLine, xLeft, yBottom, xLeft, yTop);
} // DrawOneLineSet()

//
// DrawLines -- Draw lines which have been hit.
//
public void DrawLines(Graphics g)
{
   int x, y;
   for (x = 0; x < cxWidth; x++)
   {
      for (y = 0; y < cyHeight; y++)
```

```
      {
         DrawOneLineSet(g, x, y);
      }
   }
} // DrawLines()

public void DrawDots (Graphics g)
{
   // Draw array of dots.
   int x, y;
   for (x = 0; x <= cxWidth; x++)
   {
      for (y = 0; y <= cyHeight; y++)
      {
         ptTest.X = (cxLeft - 2) + x * cxLine;
         ptTest.Y = (cyTop - 2) + y * cyLine;
         rc.Location = ptTest;
         rc.Size = szDot;
         g.FillEllipse(hbrBlack, rc);
      }
   }
} // DrawDots

public enum Side
{
   None,
   Left,
   Top,
   Right,
   Bottom
}

//
// HitTest - Check whether a point hits a line.
//
// Return values:
// 0 = miss
// 1 = hit a line
// 2 = hit and completed a square.
public int HitTest(int xIn, int yIn, int iPlayer)
{
   int x, y;
   bool bHit1 = false;
   bool bHit2 = false;
   Side sideHit = Side.None;
   for (x = 0; x < cxWidth; x++)
   {
      {         for (y = 0; y < cyHeight; y++)
```

continues

continued

```
// If already owned, do not check.
if (m_asq[x,y].iOwner != 0)
   continue;

// Otherwise check for lines against point.
if (m_asq[x,y].rcTop.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bTop) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Top;
   m_asq[x,y].bTop = true;
}
else if (m_asq[x,y].rcLeft.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bLeft) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Left;
   m_asq[x,y].bLeft = true;
}
else if (m_asq[x,y].rcRight.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bRight) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Right;
   m_asq[x,y].bRight = true;
}
else if (m_asq[x,y].rcBottom.Contains(xIn, yIn))
{
   // Line already hit?
   if (m_asq[x,y].bBottom) // Line already hit?
      return 0;
   // If not, set line as hit.
   sideHit = Side.Bottom;
   m_asq[x,y].bBottom = true;
}

// No hit in current square -- keep looking.
if (sideHit == Side.None)
   continue;

// We hit a side.
bHit1 = true;
```

```
            // Draw sides.
            Graphics g = m_ctrlParent.CreateGraphics();
            DrawOneLineSet(g, x, y);

            // Check whether square is now complete.
            // We hit a line - check for hitting a square.
            if (m_asq[x,y].bLeft &&
                  m_asq[x,y].bTop &&
                  m_asq[x,y].bRight &&
                  m_asq[x,y].bBottom)
            {
               // Side is complete.
               m_asq[x,y].iOwner = iPlayer;
               bHit2 = true;

               // Fill current square.
               FillOneSquare(g, x, y);
            }

            g.Dispose();

         } // for y
      } // for x

      if (bHit2) return 2;
      else if (bHit1) return 1;
      else return 0;

   } // HitTest

   //
   // GetScore - Get current score for player N.
   //
   public int GetScore (int iPlayer)
   {
      int iScore = 0;
      int x, y;
      for (x = 0; x < cxWidth; x++)
      {
         for (y = 0; y < cyHeight; y++)
         {
            if (m_asq[x,y].iOwner == iPlayer)
               iScore++;
         }
      }
      return iScore;
   } // GetScore

} // class Squares
```

Finally, we define two simple data structures—`Square` and `Players`—to hold details about individual game board squares and details about individual players, respectively. Listing 15.12 shows the code.

LISTING 15.12: The `Square` and `Players` Structures

```
public struct Square
{
   // Coordinate of main rectangle
   public Rectangle rcMain;
   public int iOwner;

   // Hit-rectangles of four edges of main rectangle
   public Rectangle rcTop;
   public bool bTop;
   public Rectangle rcRight;
   public bool bRight;
   public Rectangle rcBottom;
   public bool bBottom;
   public Rectangle rcLeft;
   public bool bLeft;
} // struct Square

public class Players
{
   public string strName1;
   public string strName2;
   public bool bComputerPlaying;
   public System.Drawing.Color clr1;
   public System.Drawing.Color clr2;
}
```

CONCLUSION

Whether or not you believe that good things always come in small packages, it should be clear that some very rich capabilities for creating graphical output have been placed in the very small package of the .NET Compact Framework's `System.Drawing.dll` library. This chapter looked in detail at the four types of drawing surfaces that Windows programmers are used to, two of which are supported in the .NET Compact Framework. This chapter also discussed the three families of drawing functions, which are

reasonably well represented by the .NET Compact Framework's capable, though small, feature set.

In this chapter, we explored two of the families in depth: raster and vector drawing functions. In the next chapter, we take an in-depth look at the third family of output—text. This part of the book concludes in Chapter 17 with coverage of an output device that is not officially supported in the .NET Compact Framework, namely, printers. In spite of the lack of managed-code support, we show that there are ways for a .NET Compact Framework program to satisfy the need that users sometimes have for hard copy. We do so by providing something that programmers sometimes need: some sample programs to make it clear how such an unsupported feature can, in fact, be made available.