

Chapter 15

Compact Framework Graphics

This chapter introduces the basics of creating graphical output from a Compact Framework program. [\[Comment 15.1\]](#)

An Introduction to Compact Framework Graphics	2
Drawing Surfaces	3
Display Screen	4
Printer	4
Bitmap.....	4
Metafile	5
Supported Drawing Surfaces	5
Drawing Function Families	5
Text.....	5
Raster	5
Vector [Comment 15.33]	6
Compact Framework Graphics.....	6
Role of the <code>Graphics</code> class.....	9
Drawing Support for Raster Output.....	9
Drawing Support for Vector Output	10
Drawing Support for Text Output.....	10
Drawing on the Display Screen	10
The Role of the <code>Graphics</code> class	11
Drawing in Controls	12
The <code>Paint</code> Event	14
Non- <code>Paint</code> Event Drawing	16
Raster Graphics.....	16
Specifying Colors.....	17
System Colors	17
Named Colors.....	19
Colors from RGB Values	20
Creating Brushes.....	21
Creating Brushes with System Colors.....	21
Creating Brushes with Named Colors	22
Creating Brushes with RGB Values	22
Creating Bitmaps	22
Bitmaps: drawing surface or drawing object?	23
The <code>Bitmap</code> class	24
Creating an empty bitmap	24
Creating a bitmap from an external file	25
Creating a bitmap from a resource.....	26
Image file sizes.....	29
Drawing Bitmaps	30
Drawing entire bitmap at original image size	31
Drawing part of a bitmap at original image size	31
Drawing part of a bitmap with change to image size	32
Drawing part of a bitmap with change to image size and with transparency	32
Sample: <code>ShowBitmap</code>	34

Vector Graphics.....	40
Creating Pens.....	40
A Game: JaspersDots	41
Conclusion.....	52

This chapter describes the support that a Compact Framework program has for creating graphical output. As we mention elsewhere in this book, we prefer using Compact Framework classes whenever possible. To accomplish something beyond what the Compact Framework supports, however, we drill through the managed code layer to the underlying Win32 API substrate. This chapter, and the two that follow, discusses the Compact Framework's built-in support for creating graphical output; this chapter also touches on the limitations of that support, and how to supplement that support with the help of GDI functions. [\[Comment 15.2\]](#)

An Introduction to Compact Framework Graphics

Generally speaking, programs do not create graphical output by drawing directly to device hardware¹. A program typically calls a library of graphical output functions. Those drawing functions, in turn, rely on device drivers which 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 three software layers: [\[Comment 15.3\]](#)

- 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* (`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: [\[Comment 15.4\]](#)

- 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)

.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. [\[Comment 15.5\]](#)

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`

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.

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

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. [\[Comment 15.6\]](#)

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. [\[Comment 15.7\]](#)

With just 85 of the graphical functions from the desktop's GDI library, and with none of the functions from GDI+, you might wonder whether Windows CE has enough graphic 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 the most. There are, for example, 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 can draw vector objects such as lines and polygons. [\[Comment 15.8\]](#)

For graphical output, Compact Framework programs rely on `System.Drawing.dll`, which is also the name of the graphical output library in the desktop framework. At 38K, the Compact Framework library is significantly smaller than the 456K of its counterpart on the desktop. While the desktop library supports five namespaces, the Compact Framework version supports one: `System.Drawing` (plus tiny fragments of two other namespaces). The architecture for drawing from a Compact Framework program is as follows: [\[Comment 15.9\]](#)

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

From the arrangement of these software layers, a savvy 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; (2) As on the desktop, display screens and printers require a dedicated graphic driver to operate. [\[Comment 15.10\]](#)

If possible, delegate graphical output to a control

Before you dig into Compact Framework graphics, ask yourself whether you want to create the graphical output yourself, or whether you 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. [\[Comment 15.11\]](#)

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. [\[Comment 15.12\]](#)

Sometimes, however, you do your own drawing to give your program a unique look and feel. When that is the case, you can create rich, graphical output using classes in the Compact Framework's `System.Drawing` namespace. [\[Comment 15.13\]](#)

Drawing Surfaces

On the Windows desktop, there are four types of drawing surfaces: [\[Comment 15.14\]](#)

- 1 Display Screen [\[Comment 15.15\]](#)

- 2 Printer [\[Comment 15.16\]](#)
- 3 Bitmap [\[Comment 15.17\]](#)
- 4 Metafile [\[Comment 15.18\]](#)

When we use the term 'drawing surface', we mean either a physical or 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. [\[Comment 15.19\]](#)

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 its wrapper, the `Metafile`⁴ class, does not exist in the current version of the Compact Framework. Because metafiles might someday be supported in a future version of the Compact Framework, they are worth a brief mention here. [\[Comment 15.20\]](#)

Display Screen

The display screen plays a central role in all GUI environments, for it is on the display screen where 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 very 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 touch-sensitive screen for pointing, and hardware buttons for input (supported, of course, by the on-screen keyboard. [\[Comment 15.21\]](#)

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*. [\[Comment 15.22\]](#)

Printer

Printers are the best established and most connected peripheral in the world of computers. While some industry pundits rant about the soon-to-arrive future 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 every-shrinking silicon – makes a person want something that is more real. [\[Comment 15.23\]](#)

Printing from Windows CE-powered devices is still in its infancy. Perhaps users can print what they want from desktop PCs. Or maybe the problem is that programs bundled with a Pocket PC – programs like Pocket Word and Pocket Excel – do not support printing. Whatever the cause, we show you several ways to print in chapter 18 (*Printing*), so that you can decide whether the results are worth the effort. [\[Comment 15.24\]](#)

Bitmap

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 size⁶, and treated like any other drawing surface. After a program has drawn to a bitmap, that image can be put on the display screen. [\[Comment 15.25\]](#)

If you look closely, you can see that Windows CE – and the 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 Compact Framework program. Whatever external format is used for raster data,

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.

⁶ The limitation on bitmap size is available system memory.

Windows CE prefers bitmaps. In this chapter, we show how to create a bitmap from a variety of sources and to draw those bitmaps onto the display screen from a Compact Framework program. [\[Comment 15.26\]](#)

Metafile

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* (EMF). The following Win32 native metafile functions are exported from `coredll.dll`, but are not officially supported in Windows CE. These functions might gain official support in some future version of Windows CE: [\[Comment 15.27\]](#)

- `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 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. [\[Comment 15.28\]](#)

Drawing Function Families

All of the graphical output functions can be organized into one of three drawing function families: [\[Comment 15.29\]](#)

- 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. [\[Comment 15.30\]](#)

Text

When discussing 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 is what we provide in chapter 16 (*Text and Fonts*). [\[Comment 15.31\]](#)

Raster

Raster data involves working with arrays of pixels, sometimes known as bitmaps or image data. Internally, raster data is stored as a *Device Independent Bitmap (DIB)*. As we discuss in detail later in this chapter, there are six basic DIB formats supported in the various versions of

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

Windows, including: 1, 4, 8, 16, 24, and 32 bits per pixel. Windows CE adds a seventh DIB format to this set: 2 bits per pixel. [\[Comment 15.32\]](#)

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 Compact Framework inherits these limitations, because it relies on native Win32 API functions for all of its graphic support.

Vector [\[Comment 15.33\]](#)

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. [\[Comment 15.34\]](#)

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 graphic support in the .NET Framework and in the Compact Framework, see the accompanying boxed discussion titled: *Comparing Desktop and Device Drawing Support*. [\[Comment 15.35\]](#)

Comparing Supported Desktop and Smart-Device Drawing

The *System.Drawing* namespace in the Compact Framework holds .NET elements used to draw on a device screen. While the desktop Framework provides a total of five namespaces for drawing support, in the Compact Framework this has been pared back to a lone namespace (small portions of two other namespaces are present, but mostly for the few enumerations each contains). [\[Comment 15.36\]](#)

Table 15-1 summarizing the .NET namespaces that are supported in the desktop framework, along with details of how these features are supported in the compact framework. In general, two of the six namespaces are supported. But with one – *System.Drawing.Design* – the namespace stays the same but the file it resides in changes to *System.CF.Design.dll*. The reason this changes is that this file runs on the desktop to support Visual Studio.NET, even though it is a Compact Framework-specific library. It provides information needed by the development environment to know, for example, what controls are supported in the Compact Framework and therefore which ones need to appear in the control toolbox. The change in the file name prevents collision with the desktop version, which is named *System.Design.dll*. [\[Comment 15.37\]](#)

Table 15-1 Desktop .NET Drawing Namespaces in the Compact Framework [\[Comment 15.38\]](#)

<i>Namespace</i>	<i>Description</i>	<i>Support In Compact Framework</i>
<i>System.Drawing</i>	Core drawing objects, data	A minimal set allows for the

<i>Namespace</i>	<i>Description</i>	<i>Support In Compact Framework</i>
	structures, and functions	drawing of text, raster, and vector objects with no built-in coordinate transformation.
System.Drawing.Design	Supports forms designer, and the various graphic editors of Visual Studio.NET.	Support provided by an Compact-Framework specific alternative library named System.CF.Design.dll
System.Drawing.Drawing2D	Supports advanced graphic features including blends, line caps, line joins, paths, coordinate transforms, and regions.	Not supported in CF.
System.Drawing.Imaging	Supports storage of pictures in metafiles, bitmaps, bitmap conversion, and managing metadata in image files.	Not supported in CF.
System.Drawing.Printing	Rich support for printing and the user-interface for printing.	Not supported in CF.
System.Drawing.Text	Manages fonts.	Not supported in CF.

On the surface, it would be easy to conclude that Microsoft gutted the desktop `System.Drawing.dll` library in creating the Compact Framework edition. For one thing, the desktop version is 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 framework and the compact framework – from a drawing perspective – is best appreciated by examining the `Graphics` class (a member of the `System.Drawing` namespace). The desktop framework version of this class supports 244 methods and 18 properties; the compact framework version supports only 26 methods and 2 properties. By this accounting, it appears that the prognosis of “gutted” is correct. And yet, as any thinking person knows, looks can be deceiving. [\[Comment 15.39\]](#)

To understand better the difference between desktop framework and compact framework, we are going to have to dig deeper into the `Graphics` class. To really see the differences between the desktop and compact versions, you must study the overloaded methods. If you do, you will see that the desktop framework provides many overloaded methods for each drawing call, while the compact framework provides far fewer. For example, the desktop framework provides six different ways to call `DrawString` – the text drawing function – while there is only one in the compact framework. And there are 34 versions of `DrawImage` – the function for drawing a bitmap – but only four in the compact framework. [\[Comment 15.40\]](#)

You have, in short, fewer ways to draw objects – but in general you can draw most of the same things with the compact framework that you can draw on the desktop. This is to support 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”. What you find in the compact framework implementation of the `System.Drawing` namespace is, we believe, the greatest hits of the desktop `System.Drawing` namespace. [\[Comment 15.41\]](#)

In comparing the desktop framework to the compact framework, an interesting pattern emerges that involves floating point numbers. In the desktop 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 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 framework; in the compact framework, it is the sole drawing method which accepts floating point values. [\[Comment 15.42\]](#)

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. [\[Comment 15.43\]](#)

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 framework supports seven distinct drawing coordinate systems in the `GraphicsUnit` enumeration. Among the supported coordinates systems are `Pixel`, `Inch`, and `Millimeter`. While the Compact Framework supports this same enumeration, it only has one member: `Pixel`. This means that when you draw on a device screen, you are limited to using pixel coordinates for drawing. One exception involves fonts, whose height is always specified in `Point` units. [\[Comment 15.44\]](#)

This brings up another difference between desktop framework and compact framework: available coordinate transformations. The desktop provides a rich set of coordinate transformations – scrolling, scaling, and rotating – through the `Matrix` class and the 3x3 geometric transform provided in the `System.Drawing.Drawing2D` namespace. The 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, since neither the compact framework nor the underlying operating system provides any coordinate transformation helpers. What the compact framework provides, insofar as coordinates go, is actually the same thing that the underlying Windows CE system provides, namely this: pixels, more pixels, and only pixels. [\[Comment 15.45\]](#)

While it might be lean, the set of drawing services provided in the compact framework is surprisingly complete. That is, almost anything you can draw with the desktop framework can be drawn with the compact framework. The key difference between the two implementations is that

the desktop provides a far wider array of tools and helpers for draw. Programmers of the desktop .NET framework are likely to have little trouble getting comfortable in the 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 framework code to the compact world, and are likely to have to rewrite and retrofit quite a few of their application's drawing elements. [\[Comment 15.46\]](#)

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 screen⁹, and `DrawRectangle` for drawing the outline of a rectangle. Here is a list of the other classes in the `System.Drawing` namespace for the Compact Framework: [\[Comment 15.47\]](#)

- `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 Compact Framework: `Graphics`. [\[Comment 15.48\]](#)

Drawing Support for Raster Output

Table 15-2 summarizes the methods of the `Graphics` class which draw raster data. We define raster graphics as those functions which 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. [\[Comment 15.49\]](#)

Table 15-2 `System.Drawing.Graphics` methods for Raster Drawing [\[Comment 15.50\]](#)

Method	Comment
<code>Clear</code>	Accepts a colors value, and uses that value to fill the entire surface of a window or the entire surface of a bitmap.
<code>DrawIcon</code>	Draws an icon at a specified location. An icon is a raster image created from two rectangular bitmap masks. The <code>DrawIcon</code> function 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

⁸ Fully-qualified name: `System.Drawing.Graphics`.

⁹ `DrawImage` can also be used to draw bitmaps onto other bitmaps.

Method	Comment
	also slower to draw.
DrawImage	Draws a bitmap onto the display screen, or draws a bitmap onto the surface of another bitmap.
FillRegion	Fill 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.

Drawing Support for Vector Output

Table 15-3 summarizes the seven methods in the `Graphics` class which draw vector graphic objects. The set of supported vector methods are quite a bit fewer than what you find 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. [\[Comment 15.51\]](#)

Table 15-3 `System.Drawing.Graphics` methods for Vector Drawing [\[Comment 15.52\]](#)

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

Drawing Support for Text Output

Table 15-4 summarizes the methods of the `Graphics` class which 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 object on a sea of pixels. When dealing with a lot of text, it is important to measure the size of each text box to make sure that 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 your program's users. Even if a user does not immediately notice minor spacing problems, the human eye is very finicky insofar as what makes for acceptable text. Poor spacing makes text harder to read because a reader must strain their eyes to read the text. Properly-spaced text makes readers – and their eyes – happier than poorly-spaced text. [\[Comment 15.53\]](#)

Table 15-4 `System.Drawing.Graphics` methods for Text Drawing [\[Comment 15.54\]](#)

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

Drawing on the Display Screen

The various `System.Drawing` classes in the 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. [\[Comment 15.55\]](#)

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, which includes

functions to draw icons, functions to draw bitmaps, and also functions that fill regions¹⁰ or the entire display screen. The third family of graphical functions, vector functions, support the drawing of lines, polygons, rectangles, and ellipses on the display screen. [\[Comment 15.56\]](#)

The Role of the `Graphics` class

For a Compact Framework program to draw on the display screen, it must have an instance of this class. And so you might wonder how to get access to an instance of the `Graphics` class? 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. The only reasonable conclusion is that there is some other way to instantiate the `Graphics` class. [\[Comment 15.57\]](#)

Close study of the 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 which are needed to gain access to a graphics object. We 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. [\[Comment 15.58\]](#)

Table 15-5 Compact Framework methods for accessing a `Graphics` object [\[Comment 15.59\]](#)

Namespace	Class	Method	Comment
<code>System.Drawing</code>	<code>Graphics</code>	<code>FromImage</code>	Creates a graphic object for drawing onto a bitmap. When done drawing, clean up the graphic object by calling <code>Dispose</code> .
	<code>Graphics</code>	<code>Dispose</code>	Reclaim memory used by graphic objects
<code>System.Windows.Forms.</code>	<code>Control</code>	<code>CreateGraphics</code>	Create a graphic 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 graphic object by calling <code>Dispose</code> .
	<code>Control</code>	(Paint Event Handler)	Obtain a graphic object to handle a paint event. As indicated in table 15-6, only five control classes support this event. Do not call <code>Dispose</code> when done drawing.

The display screen is a shared resource. A multi-tasking, multi-threaded 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 only allowed in a window (that is, in a form or a control). [\[Comment 15.60\]](#)

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 which derive from the `Control` class. [\[Comment 15.61\]](#)

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 Compact Framework needs to recycle the contents of that graphics object for other controls to use¹¹. [\[Comment 15.62\]](#)

¹⁰ A region is a set of rectangles. Regions exist primarily in support of clipping, but can also be used for drawing into.

¹¹ And ultimately, the window manager reuses the device context contained within the graphics object.

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 graphic object, which is not happens. Instead, like the graphic object that arrives with the paint event, the graphic object that is provided by the `CreateGraphics` method is loaned to you from a supply that is created and owned by the Windows CE window manager. Therefore, you are required to return this object when you are done by calling the graphic object's `Dispose` method. Failure to make this call results in a program hanging. [\[Comment 15.63\]](#)

Calling Dispose for a Graphics Object

There are two ways to get a graphics object, but you only need to call `Dispose` for one of those ways. You must call the `Dispose` method for graphic objects which are returned by the `CreateGraphics` method. But you do not call `Dispose` for graphic objects which are provided as a parameter to the paint event handler. [\[Comment 15.64\]](#)

The third way to get a `Graphics` object is by calling the `shared` `FromImage` method in the `Graphics` class. On the desktop, the `Image` class is a `MustInherit` class which serves as the base class for the `Bitmap` and `Metafile` classes. Since metafiles are not supported in Compact Framework, the `FromImage` method can only return 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, and instead direct your attention to the subject of drawing in controls. [\[Comment 15.65\]](#)

As we discuss in chapter 7 (*Inside Controls*), the watchword for Compact Framework controls is that "inherited does not mean supported." Of the 28 available Compact Framework control classes, only five support drawing. To help understand what drawing is supported, we start by identifying the specific controls that you can draw onto. We are then going to cover the most important control event for drawing, the `Paint` event. We then discuss how non-paint event drawing differs from paint event handling. [\[Comment 15.66\]](#)

Drawing in Controls

In the Desktop 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 owner-draw. In other words, you can get a `Graphics` object for every type of control¹², and use that object to draw inside the client area of any control. Owner-draw support is widely supported 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. [\[Comment 15.67\]](#)

Things are different in the Compact Framework, for reasons which are directly attributable to the Compact Framework design goals. As we discuss in chapter 7, *Inside Controls*, the Compact Framework itself was built to be as small as possible, and also to allow Compact Framework programs to run with reasonable performance. The result is a set of controls¹³ with the following qualities: [\[Comment 15.68\]](#)

¹² All desktop control classes that we tested support the `CreateGraphics` method. However, a few desktop controls classes do not support the overriding of the `Paint` event: `ComboBox`, `HScrollBar`, `ListBox`, `ListView`, `ProgressBar`, `StatusBar`, `TabControl`, `TextBox`, `ToolBar`, `TrackBar`, `TreeView`, and `VScrollBar` classes.

¹³ We discuss this point in detail in chapter 7 (*Inside Controls*).

- Compact Framework controls rely heavily on the built-in, Win32 API control classes.
- Compact Framework controls do not support every property, method, and event inherited from the base `Control`¹⁴ class.

The result is that only a few 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 by Compact Framework control classes. [\[Comment 15.69\]](#)

Table 15-6 Support for Drawing in Compact Framework control classes [\[Comment 15.70\]](#)

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

As suggested by the column headings in table 15-6, there are two types of drawing: paint-event drawing and `CreateGraphics` 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 controls in table 15-6 support paint drawing. We turn our attention now to the subject of the paint event, and its role in the Windows CE user interface. [\[Comment 15.71\]](#)

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? [\[Comment 15.72\]](#)

If you are willing to think outside of the managed-code box, and perhaps to step outside of it, you can draw on any control at any time. The Compact Framework team did a great job at 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 Compact Framework team that you should not be allowed to draw inside controls. [\[Comment 15.73\]](#)

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` to retrieve a DC handle, and `ReleaseDC` when you are done. [\[Comment 15.74\]](#)

A second method for accessing a device context is by this call: `hdc = CreateDC(NULL, NULL, NULL, NULL)`. The DC that is returned provides access to the entire display screen, not just inside windows. Among its

¹⁴ Fully-qualified name: `System.Windows.Forms.Control`.

other uses, this type of DC is useful for taking screen shots of the display screen, which can be useful for creating documentation. When done with the DC, be sure to clean up after yourself by calling the `DeleteDC` function. [\[Comment 15.75\]](#)

The `hdc` that is 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 `ReleaseDC` or `DeleteDC`. [\[Comment 15.76\]](#)

The Paint Event

To draw in a window – that is, in a form or in a control – you handle the `Paint`¹⁵ 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. [\[Comment 15.77\]](#)

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 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`¹⁶ class.) But what causes a window to become invalid? There are several causes. [\[Comment 15.78\]](#)

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). [\[Comment 15.79\]](#)

A window can also get 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. [\[Comment 15.80\]](#)

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, or 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 is what must be drawn in response to a paint event. [\[Comment 15.81\]](#)

The final mechanism that triggers a paint event is that 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

¹⁵ What Win32 and MFC programmers know as a `WM_PAINT` message.

¹⁶ Fully-qualified name: `System.Windows.Forms.Control`.

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. [\[Comment 15.82\]](#)

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, for the entire window, and optionally allows you to request that the background be erased prior to the paint event. [\[Comment 15.83\]](#)

This approach to graphical window drawing is not new to the 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 on up to the current GUI systems shipping today. A window holds some data, and displays a view of that data. [\[Comment 15.84\]](#)

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. The two basic problems that can be observed with the paint event are as follows: [\[Comment 15.85\]](#)

- 1 Failure requesting paint (cold window with stale contents)
- 2 Requesting paint events too often (hot window flickers, annoying 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 much. What is needed is the happy medium; to invalidate a window the right number of times, and at just the right times. [\[Comment 15.86\]](#)

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 Compact Framework controls. Here is an empty paint event handler: [\[Comment 15.87\]](#)

```
Private Sub FormMain_Paint(
    ByVal sender As Object, _
    ByVal e As PaintEventArgs) Handles MyBase.Paint

    Dim g As Graphics = e.Graphics
    ' ...draw...

End Sub
```

The second parameter to the paint handler is an instance of a `PaintEventArgs`¹⁷. 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. [\[Comment 15.88\]](#)

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. For windows which contain more dynamic data, drawing outside the confines of the paint event might be desirable. [\[Comment 15.89\]](#)

¹⁷ Fully-qualified name: `System.Windows.Forms.PaintEventArgs`.

Non-Paint Event Drawing

A window that contains any graphical output must handle the paint event. A window whose contents must change quickly might need to draw in response to events other than the paint event. A program which is displaying some type of animation, for example, might draw in response to a timer event. [\[Comment 15.90\]](#)

A program that echoes user input might draw in response to keyboard or mouse events. Figure 15-1 shows the `DrawRectangles` program, a sample for chapter 6 (*Mouse and Keyboard Input*). This program draws rectangles in the program's main form, using a pair of (x,y) coordinates. One coordinate pair is collected for the mouse-down event, and a second coordinate pair is collected for the mouse-up event. As the user is moving the mouse (or a stylus on a Pocket PC), the program draws a stretchable rubber rectangle as the mouse/stylus is moved from the mouse-down point to the mouse-up point. The program accumulates rectangles as the user draws them. [\[Comment 15.91\]](#)

The `DrawRectangles` program uses both paint and non-paint drawing. In response to the paint event, the program draws each of the accumulated rectangles. In response to the mouse move event, the stretchable rectangle is drawn to allow the user to preview the location of a mouse before committing to a specific rectangle location. [\[Comment 15.92\]](#)

The basic template for the code used in non-paint drawing appears here:

```
Dim g As Graphics = CreateGraphics()  
' ...draw...  
g.Dispose()
```

This follows a programming pattern familiar to some as the *Windows sandwich*¹⁸. 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 consist of the drawing, which is accomplished with the drawing methods from the `Graphics` class. [\[Comment 15.93\]](#)

Raster Graphics

We define raster graphics as those functions which operate on an array of pixels. The simplest raster operation is to fill a rectangle with a single color. 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 single color, often white or sometimes gray. Three methods in the `Graphics` class can be used to fill a rectangular area: [\[Comment 15.94\]](#)

- `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`¹⁹. The other two methods accept a `Brush`²⁰ 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. [\[Comment 15.95\]](#)

¹⁸ This is what Eric Maffei of MSDN Magazine used to refer to as the Windows Hoagie.

¹⁹ The fully-qualified name for this value type is: `System.Drawing.Color`.

²⁰ Fully-qualified name: `System.Drawing.Brush`.

Thank You

Thank you for taking the time to read this preview chapter. We hope it has provided you insights and tips to help with your Compact Framework programming project. You can help us create a better book by clicking the comment link at the end of each paragraph, and sending your comments and suggestions on our review web site.

Preview Chapter Text

Our public review site provides the complete table of contents for the Compact Framework book at this link: <http://www.paul Yao.com/cfbook.htm>. That table of contents contains links to all the preview chapters. The preview chapter provides the complete outline of topics covered in a chapter, and also the first section or two from each chapter.

Complete Chapter Text

You can get the complete text for each chapter, available to readers who register at our web site. Registration is simply and easy – we only ask for an email address. To register, click on this link: <http://www.paul Yao.com/ReaderFeedback/Logon.aspx>.

When you register, you can download the available chapters from the full-text Table of Contents, available at this link: <http://www.paul Yao.com/ReaderFeedback/default.aspx>. We notify registered readers of new chapters – and chapter updates – as they become available.