GUI applications revolve around events, and F# provides a natural way to process events with functions. Graphical interfaces are often developed using visual editors, in particular to build GUls by assembling controls. Applications, however, often need drawing capabilities for displaying and manipulating data, which requires custom programming to augment available controls. This chapter discusses how to develop graphical applications with F# and why the functional traits of the language suit the event-driven programming paradigm typical of GUls.

In this chapter, you use the Windows Forms library, the graphic toolkit designed by Microsoft on top of GDI+ and Win32 that is the former Windows presentation system. The Windows Presentation Foundation (WPF) is the new presentation framework for the Windows platform; it’s discussed later in this chapter.

You may wonder why it’s useful to know about the old-fashioned Windows Forms toolkit. The answer is twofold: on one hand, the framework structure is shaped after a consolidated model that dominated graphical programming for decades and is still used in many other frameworks for programming GUls such as GTK#, a .NET managed library for writing applications based on the GTK toolkit; on the other hand, Windows Forms is still the GUI toolkit for programming managed applications for mobile devices based on Windows Mobile.

Writing “Hello, World!” in a Click

It’s traditional to start with a “Hello, World!” application, so let’s honor that and begin with a simple program that provides a button to display the magic phrase when clicked:

```fsharp
open System.Windows.Forms

let form = new Form(Text="Hello World WinForms")
let button = new Button(Text="Click Me!", Dock=DockStyle.Fill)

button.Click.Add(fun _ -> MessageBox.Show("Hello, World!", "Hey!") |> ignore)
form.Controls.Add(button)
form.Show()
```

Even in its simplicity, the application captures many traits typical of GUI applications. After opening the namespace associated with Windows Forms, you create the form `form` that contains the button `button`, set the form and button captions by assigning their `Text` properties, and tell the button that it should fill the entire form.
Most of GUI programming is devoted to handling events through callbacks from the graphical interface. Events are described in Chapter 8. To display a message box containing the "Hello, World!" string, you have to configure the button so that when its Click event is fired, a function is called. In the example, you pass a function to the Add method for the button's Click event, which adds an event handler to an event source. You then add the button to the form and call the form's Show method to display it.

Note that this code should be executed using fsi.exe. It won’t run as a stand-alone application unless you add the following line at the end:

Application.Run(form)

This line relates to the event loop of a GUI application, and it’s required to handle events such as button clicks. Moreover, if you execute the compiled program, notice that the window uses the classic Windows look and feel rather than the more fashionable look and feels featured by recent versions of Windows. This can be easily addressed by adding the following call to the EnableVisualStyles static method, right after the open statement:

Application.EnableVisualStyles()

If you use fsi.exe, both visual styles and the event loop are handled by F# Interactive.

Understanding the Anatomy of a Graphical Application

Graphical applications are built on the abstractions provided by the graphical environment hosting them. The application must interact with its environment and process input in an unstructured way. User input isn’t the only kind of input received from a windowing system. Window management often involves requests to or from the application itself, such as painting or erasing a form.

Windowing systems provide a common and abstract way to interact with a graphical application: the notion of an event. When an event occurs, the application receives a message in the message queue with information about the event.

The graphical application is responsible for delivering messages from the message queue to the control for which they’re meant. A set of functions provided by the API of the windowing system supports this. This activity of reading messages from the message queue and dispatching them is known as the event loop of the application. If the loop fails for any reason, the GUI components cease to work, the application hangs, and Windows may eventually inform you that the application isn’t responding.

It’s rare for an application to program the event loop explicitly. Programming toolkits encapsulate this functionality because it’s basically always the same. The Run method of the Application class is responsible for handling the event loop, and it ensures that messages related to events are delivered to targets within the application.

GUI programs often involve multiple threads of execution. Chapter 13 discusses threading and concurrency in more detail; for this chapter, it’s important to remember that event dispatching is a single-threaded activity, even if it may seem the opposite. The thread executing the event loop calls the functions and methods registered for handling the various events. In the “Hello, World!” example, for instance, you told the button to call back the function to show the message box when clicked.