.NET GUI Programming

Rex Jaeschke

Edition: 1.0
Printing: October 9, 2003

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
rex@RexJaeschke.com

This book was typeset by the author using the TEX typesetting package, LATEX macros, and Makeindex indexing tool. TEX is a trademark of the American Mathematical Society.

Contents

Preface

This text introduces Graphical User Interface (GUI) programming in a .NET environment. In general, the material is not hardware or operating system-specific; however, the GUI builder used is that provided in Visual Studio .NET.

Reader Assumptions

To fully understand and exploit the material, you should be conversant with the following concepts and the syntax required to express them in C#, J#,[1], or Visual Basic:

- Basic Language Elements
- Looping and Testing
- Methods
- References, Strings, and Arrays
- Classes
- Inheritance
- Interfaces
- Exception handling
- Operator Overloading
- Delegates
- Structs
- Namespaces
- Programs, Assemblies, and Accessibility
- Input and Output
- The Preprocessor

It would be advantageous if you have also sucessfully completed my 1-day seminar *Visual Studio .NET*, or have equivalent experience. That seminar allows attendees to

- Navigate their way around most of the menu options.
- Be able to modify and create toolbars.
- Understand docking of windows.
- Have considerable mastery of the text editor with respect to searching, copying, and replacing text, both in memory and in one or more disk files, as well as programming language-specific support.
- Understand solutions and projects, and how to create them.
- Understand properties, and how to inspect and set them.
- Know how to use the source code browser.
- Know how to use the debugger by setting breakpoints, tracing functions calls, examining and changing variables, among other things.

Prior experience in GUI programming is not required.

[1] J# is a language that is a superset of Java. The extensions it contains allow programs written in it to interact with, and fully exploit the .NET environment.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training courses for some 12 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and well-focused. Many books on object-oriented languages use objects, inheritance, exception handling, GUI, and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other; I simply know that my approach works well, and has formed the basis of my successful seminar business for more than a decade.

Finding the Source Code for Your Language

This book is intended to be used by programmers who know C#, J#, or Visual Basic .NET. Ordinarily, it is advantageous to have the source code shown in-line, at the point of its description. I have done this in the first chapter, since there are numerous direct references to the code; however, that requires the reader to skip over the code for the other languages. In subsequent chapters, there are few direct references to code, and instead of showing all languages in-line, references are made to the corresponding place in an appendix where they can be found. For example, at the end of the discussion of the example Cntb01 in chapter 2, the following text is present:

> The source code for example Cntb01 can be found on pages *x* and *y* (VB).

The programs shown in or referred to the text are available electronically in a directory tree named Source. This directory contains a number of subdirectories, one each for source language. Within each of those each chapter has its own subdirectory.

Exercises and Solutions

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, in which the subdirectory structure is as for Source. Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See lab directory *xx*.)" This indicates the corresponding solution in the Labs directory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Future of .NET

Microsoft announced the C# language and .NET platform in July, 2000. At that time it was stated that both the language and some subset of the library and runtime environment would be submitted for standardization, to ECMA, an international standards organization. In September, 2000, ECMA committee TC39, which was responsible for standardization of the scripting language ECMAScript (known commercially as JavaScript and JScript), agreed to take on this new work. As a result, TC39 was split into three Task Groups: TG1 – ECMAScript, TG2 – C#, and TG3 – Common Library Infrastructure (CLI). I serve as project editor of both TG2 and TG3.

The submission to ECMA was sponsored by Hewlett-Packard, Intel, and Microsoft. A number of other companies also agreed to participate in the standards work, which began in earnest, in November, 2000. This work was completed in September, 2001, and both standards were adopted by ECMA in December, 2001. The standard for CLI is ECMA-335, and that for C# is ECMA-334. (These standards can be downloaded free of charge from www.ecma.ch.)

In 2002, these standards were submitted to ISO/IEC JTC 1 for its consideration for adoption as "ISO Standards." This process was finalized early in 2003, and the resulting standards are ISO/IEC 23271 (CLI) and ISO/IEC 23270 (C#). A CLI-related Technical Report is available as ISO/IEC 23272. (Information on these standards can be obtained from www.iso.ch.)

Rex Jaeschke, January, 2003

Chapter 1

The Basics

In this chapter, we will be introduced to forms, events, and controls.

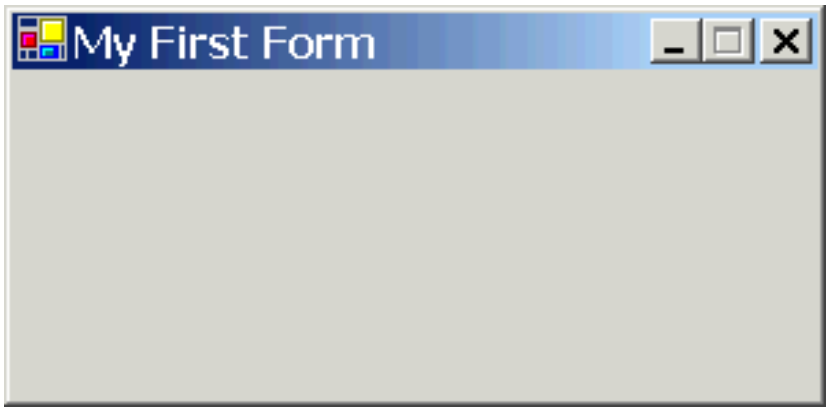
1.1 Introduction

The .NET library contains numerous classes that can be used to create Graphical User Interfaces (GUIs). A GUI is made up of a *form* upon which are placed one or more *controls*, such as buttons, checkboxes, and scrollbars. When we interact with a control by pressing its button, checking its checkbox, or moving its scrollbox, for example, that control generates an *event*, which can be detected and serviced by an *event handler*. All controls are rectangular in shape and have a horizontal or vertical orientation. A form is a control.

While a control is often a GUI object, some controls contain other controls, in which case, they are referred to as *containers*. A container can contain controls as well as other containers. A *top-level container*, such as a form, cannot be part of any other container. Every container is a control, but a control is not necessarily a container.

1.2 Forms

Let's begin by looking at the following simple form (see directory Ba01):

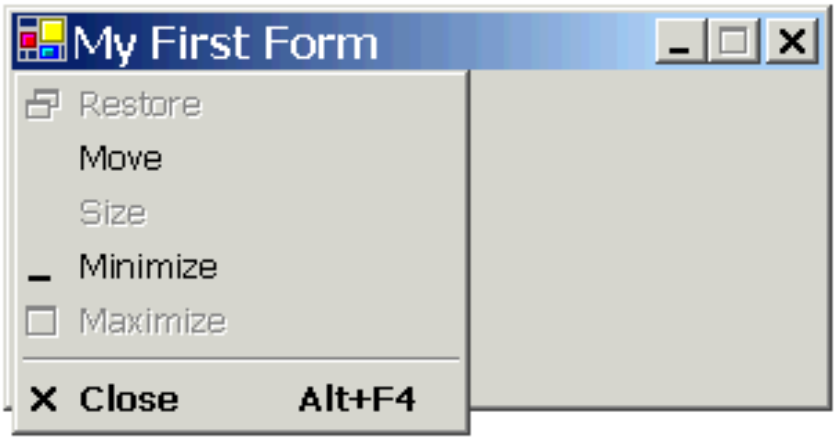


This form has a *titlebar* across its top, and that titlebar contains the following elements, from left to right:

- An icon.
- A title, in this case "My First Form".
- A Minimize button.
- A Maximize button (that is disabled).
- A Close button.

The form contains no controls and no menu, and its size is fixed (that is, it cannot be resized by dragging its edges or corners).

When the icon is clicked, the menu shown below appears, allowing us to select any of the options highlighted:



Selecting Minimize corresponds to clicking the Minimize button on the right; both result in the form's being reduced to an entry on the taskbar. Assuming it were enabled, selecting Maximize would correspond to clicking the Maximize button on the right; both result in the form's being enlarged to take up the whole screen. Once a form is maximized, this button changes to a pair of overlapping windows.

The application can be terminated in one of four ways: by clicking on the Close button (showing an X), by double-clicking on the icon, by selecting the icon and choosing Close, or by pressing Alt+F4.

Here is the C# source code used to make this form:

```
using System.Windows.Forms;

/*1*/
public class MyForm : Form
{
        public MyForm()
        {
/*2*/             this.Text = "My First Form";
/*3*/             MaximizeBox = false;
/*4*/             FormBorderStyle = FormBorderStyle.FixedSingle;
/*5*/             Height = 150;
/*6*/             Width = 300;
/*7*/             StartPosition = FormStartPosition.CenterScreen;
        }

        public static void Main()
        {
/*8*/             Application.Run(new MyForm());
        }
}
```

and here is the equivalent Visual Basic .NET source code:

```
Imports System.Windows.Forms ' redundant when using the GUI builder

'1
Public Class MyForm
        Inherits Form
        Public Sub New()
'2
                Me.Text = "My First Form"
'3
                MaximizeBox = False
'4
                FormBorderStyle = FormBorderStyle.FixedSingle
'5
                Height = 150
'6
                Width = 300
'7
                StartPosition = FormStartPosition.CenterScreen
        End Sub
End Class
```

All forms are derived from class `Form`, which is in namespace `System.Windows.Forms`.

When the C# version of the application is started in case 8, the `Run` method of class `Application` is called. According to the on-line help, this class "Provides static methods and properties to manage an application, such as methods to start and stop an application, to process Windows messages, and properties to get information about an application." There are a number of overloads for `Run`. The one used here starts the application and displays the specified form, an instance of class `MyForm`. As shown in cases 2–7, the constructor for this instance sets a number of the object's properties to specific values. All properties not given explicit values take on their default values.

> **Visual Basic note:** The entry-point method `Main` is not needed. Instead, we need to change the startup object in the project's properties to `MyForm`.

In case 2, the `Text` property is explicitly shown as belonging to the current object. Since this would be the default lookup, the explicit prefix is omitted in the subsequent statements. By default, `Text` is an empty string.

By default, the Minimize, Maximize, and Close buttons are enabled. In case 3, the Maximize button is disabled, which is reflected by its being "greyed out" in the titlebar and icon menu.

The border of a form and certain aspects of its titlebar are controlled via the property `FormBorderStyle`, whose value can be any member of the enumeration type of the same name. In case 4, the style `FixedSingle` has been chosen, which results in a single-line border and a form that cannot be resized. (The default style is `Sizeable`.)

By default, the size of a form is 300x300 pixels. The `Height` and `Width` properties are set in cases 5 and 6, respectively, although case 6 really is redundant.

The position of the form on the screen is controlled by the property `StartPosition`, whose value can be any one of the members of the enumeration `FormStartPosition`. The default value is `WindowsDefaultLocation`; this is overridden in case 7 by `CenterScreen`, which has the obvious effect.

The icon used in the titlebar is the default as used by the GUI builder in Visual Studio .NET. This can be overridden via the property `Icon`.

Languages that do not support properties directly (such as J#) must use method calls instead to set and/or get property values. .NET supports this by treating methods whose names are of the form `set_`*property* and `get_`*property* as property value setters and getters, respectively. For example, here is the J# code for setting the properties in Ba01:

```
/*2*/          this.set_Text("My First Form");
/*3*/          set_MaximizeBox(false);
/*4*/          set_FormBorderStyle(FormBorderStyle.FixedSingle);
/*5*/          set_Height(150);
/*6*/          set_Width(300);
/*7*/          set_StartPosition(FormStartPosition.CenterScreen);
```

Example Ba01 shows the application's entry point, `Main`, as being part of the form class. (This is also the approach used by the code generator in Visual Studio .NET's GUI builder.) However, for C# and J#, the startup method can just as easily be in a different class—and source file as well. (An example of this can be seen in directory Ba02.)

Let's use the Visual Studio .NET GUI builder to create a form (and corresponding source code) equivalent to that produced by hand using Ba01. The steps involved are as follows:

1. Create a C#, J#, or Visual Basic project called Ba03 using the Windows Application template.
2. In the Solution Explorer, rename the source file Form1 to Ba03.
3. Right-click on the form picture, and select Properties.
4. Set the form's name (via the property `(Name)`) to `MyForm`.
5. Set the `Text` property to "My First Form", and the form title changes.
6. Set the `MaximizeBox` property to `false`, and the Maximize button is greyed out.
7. Set the `FormBorderStyle` property to `FixedSingle`.
8. The `Size` property is displayed as a height/width pair. Expand this entry and set `Height` to 150 pixels, noting that `Width` is already 300. `Size` now changes to `300, 150`. (`Size` is an instance of the struct type `Size`, which contains an integer pair. Under program control, we can either set `Size` to a new size value pair, or we can set the size's properties `Height` and `Width` individually.)
9. Set the `StartPosition` property to `CenterScreen`.
10. Right-click on the form, and select View Code to see the source code generated. Replace all occurrences of "Form1" with "MyForm". (Depending on certain circumstances, this change may not be necessary. If such replacement results in zero changes, that just means this change isn't necessary.)
11. **Visual Basic note:** change the startup object in the project's properties to `MyForm`.

    Exercise 1-1: Create Ba03 as directed above, then build and run the application, making sure it behaves as discussed earlier. Specifically:

    1. Make sure the form starts out in the center of the screen, and that it has the correct size.
    2. Check that the Maximize option is disabled both on the button and the icon's menu, and that the form cannot be resized
    3. Minimize the form.
    4. Try all four ways to close the form.
    5. When either of the `MinimizeBox` or `MaximizeBox` properties is enabled and the other is disabled, the disabled one is greyed out on the form. What happens if they are both disabled?
    6. What happens when you set the `ControlBox` property to False? (To get a brief description of the purpose of a property, from its right-click menu select Description. Doing so for one property causes descriptions to be displayed at the bottom of the Properties Window for all subsequent properties selected, not just that one from whose right-menu you selected.)
    7. Experiment with various FormBorderStyles including `Sizeable`, reading the on-line documentation for each.
    8. Experiment with various StartPositions, reading the on-line documentation for each.
    9. To revert to the default value for any property, from its right-click menu, select Reset. Do this for `Size` and then set `Height` back to 150. Note that the font of the values of properties that are not the default, is bold.
    10. Look briefly at the descriptions of all other form properties, just to get an overview of all of the characteristics of a form that can be set via the Properties Window and/or under program control.
    11. Look at the tooltips for each of the buttons on the toolbar at the top of the Properties Window. Select each one in turn.

12. Look at the on-line documentation for class `Form` to see the instance methods it defines, as well as those it inherits from its (indirect) base class `Control`.

Let's look at the source code generated by the GUI builder (see directory Ba03):

```
C# code

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

The code generator provides access to a number of namespaces, even though the code generated only needs `System`, `System.Drawing`, and `System.Windows.Forms`. Since there is no harm in this, and code added by a programmer in the future may well use these other namespaces, there is no need to remove them.

> **Visual Basic note:** When using Visual Studio to compile code, a number of namespaces are automatically imported by the IDE. These include Microsoft.VisualBasic, System, System.Collections, System.Data, System.Diagnostics, System.Drawing, and System.Windows.Forms.

```
C# code

namespace Ba03
{
        public class MyForm : System.Windows.Forms.Form
        {
                private System.ComponentModel.Container components = null;

VB code

Public Class MyForm
        Inherits System.Windows.Forms.Form

        Private components As System.ComponentModel.Container
```

The C# namespace name is taken from the project name, in this case, `Ba03`. (This is also true for the package name in J#. In the case of Visual Basic, the namespace is taken from the project's Root namespace property.) Although in many cases it is not necessary to use namespaces, doing so doesn't hurt either, and it can get us into a programming-in-the-large mindset. So we'll keep that.

As we can see, the form class `MyForm` is derived from `System.Windows.Forms.Form`. The form is a container; however, since it contains no controls, `components` is set to `null` (or `Nothing` for VB).

```
C# code

                public MyForm() ...

                protected override void Dispose( bool disposing ) ...

//              Windows Form Designer generated code ...

VB code

                Public Sub New() ...

                Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean) ...

'               Windows Form Designer generated code ...
```

A no-argument constructor is defined for the class. Note that this line (as well as several others) ends in ... to show that it is a Visual Studio .NET outlining place holder. (We'll look at the expanded contents of these hidden blocks later.) Other constructors can be added manually, as necessary.

A method called `Dispose` is also defined. This is called to clean-up the object before it is garbage-collected.

```
C# code
                [STAThread]
                static void Main()
                {
                        Application.Run(new MyForm());
                }
        }
}
```

Finally, for the C# version (as well as for the J# version, whose source is not shown here), the entry-point method `Main` is defined, and it has the attribute `STAThreadAttribute`. This piece of magic is used to set the default threading model for the application. (If neither it nor its sibling `MTAThreadAttribute` is present, the main thread is not initialized "correctly".) We won't discuss it further here other than to say that this is generated automatically, and we'll keep it.

Here's the complete source for the no-argument constructor:

```
C# code

public MyForm()
{
        InitializeComponent();
}

VB code

Public Sub New()
        MyBase.New()
        InitializeComponent()
End Sub
```

Rather than putting all the initialization code directly in the constructor, the generator puts it in a private method called `InitializeComponent`, so that text can be hidden using outlining, leaving only the programmer-added code.

Here is the Windows Form Designer generated code:

```
C# code

#region Windows Form Designer generated code
private void InitializeComponent()
{
        //
        // MyForm
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(294, 122);
        this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle;
        this.MaximizeBox = false;
        this.Name = "MyForm";
        this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;
        this.Text = "My First Form";
}
#endregion
```

```
VB code

#Region " Windows Form Designer generated code "
Private Sub InitializeComponent()
        '
        'MyForm
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(294, 122)
        Me.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedSingle
        Me.MaximizeBox = False
        Me.Name = "MyForm"
        Me.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen
        Me.Text = "My First Form"
End Sub
#End Region
```

This looks much just like the code we saw in Ba01, except that the height and width properties are not set directly, but are achieved by setting several other properties instead.

For completeness, here is the source for `Dispose`:

```
C# code

protected override void Dispose( bool disposing )
{
        if( disposing )
        {
                if (components != null)
                {
                        components.Dispose();
                }
        }
        base.Dispose( disposing );
}

VB code

Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
                If Not (components Is Nothing) Then
                        components.Dispose()
                End If
        End If
        MyBase.Dispose(disposing)
End Sub
```

Well, we've "written" our first GUI application; however, we did it without actually writing any code, and the approach was the same for all the languages.

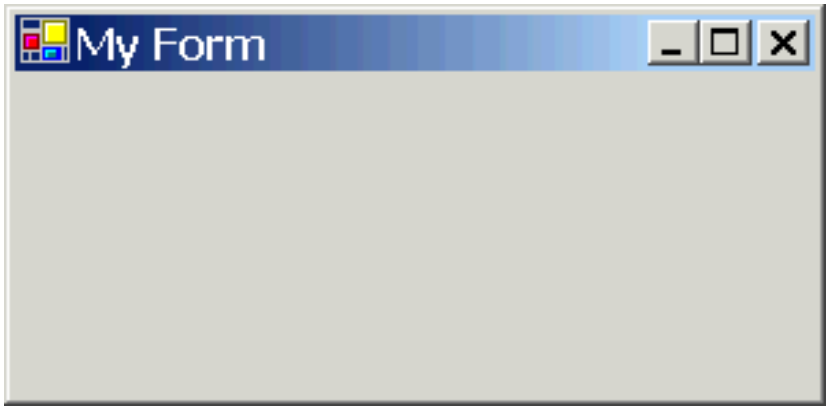
1.3 Events

When a user interacts with a control (remember, a form is a control), an event is generated, and the handler for that event type on that control is called. If there is no handler for that event type on that control, the interaction is ignored.

.NET allows multiple handlers to be registered for the same event type on the same control. It does this via delegates. Remember, a delegate is an object that can encapsulate one or more methods having the same signature. Once a delegate has been made to encapsulate a set of methods, those methods can all be invoked via that delegate,

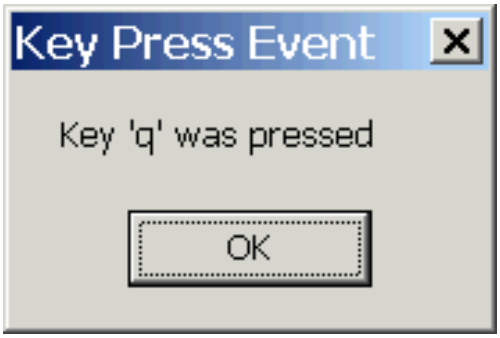
without the caller knowing which methods have been encapsulated. Methods can be added to or removed from a delegate under program control.

Consider the following (very basic) form (see directory Ba04):



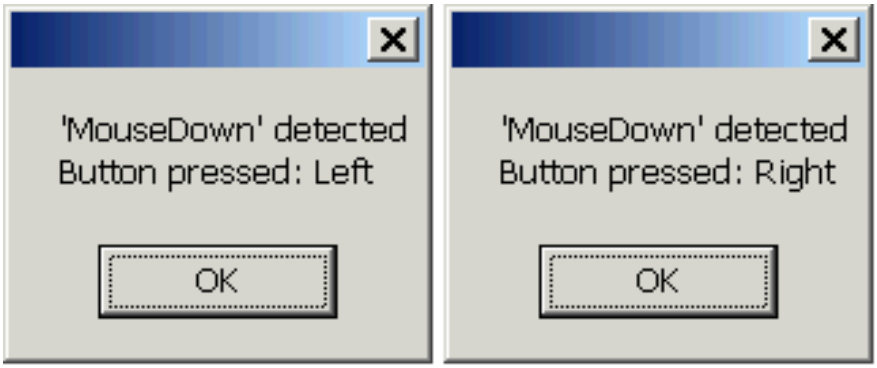
It's 300x150 pixels in size, it's sizeable, and it has all its titlebar buttons active.

Consider the case in which we wish to have the form detect a key press when that form has the focus. For example, when the 'q' key is pressed, we want a *MessageBox* like the following to be displayed:

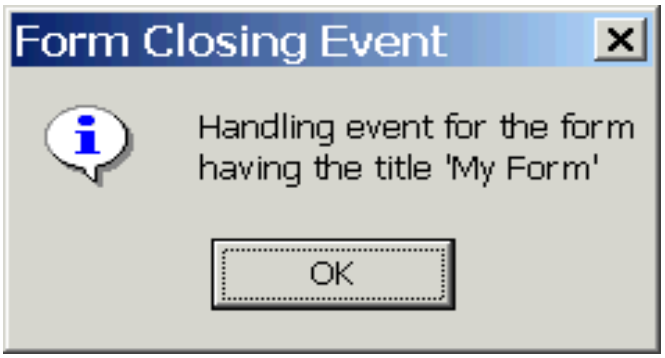


A MessageBox is a predefined dialog box that displays application-related information to the user. (MessageBoxes can also be used to get information from the user.)

We also wish to have the form detect a mouse click when that form has the focus. For example, when the left or right button is pressed, we want the corresponding MessageBox to be displayed, as follows:



Finally, we want the program to get control when the form is closed in any of the four ways. For example, we want the following MessageBox to be displayed:



This requires that we define a separate handler for each of these event types. Here are the steps used with the Visual Studio .NET GUI builder for application Ba04:

1. Create a C#, J#, or Visual Basic project called Ba04 using the Windows Application template.
2. Display the form's Properties Window, set the `Text` property to 'My Form', set the `Height` property to 150 pixels, and set the `StartPosition` to `CenterScreen`.
3. Press the Events button in the Properties Window to display the set of possible events that can be detected by a form. Press the Alphabetic button to get them sorted in that order.

   Visual Basic note: There is no Events button in the Properties Window. Instead, we must go to the document window and make the source code file active. Across the top of this window are two drop-down boxes whose ToolTips read, from left-to-right, Class Name and Method Name, respectively. In the class name list, select (Base Class Events). Now the method name list contains a set of event types.

4. For the event type `Closing`, enter a value of `Form1_Closing`. This is the name of the method that will handle that type of event on the form. By convention, its name is the form (or control) type, followed by an underscore, followed by the exact name of the event type. (In the case of Visual Basic, this name is generated automatically.)
5. For the event type `KeyPress`, enter a value of `Form1_KeyPress`. (In the case of Visual Basic, this name is generated automatically.)
6. For the event type `MouseDown`, enter a value of `Form1_MouseDown`. (In the case of Visual Basic, this name is generated automatically.)

Apart from the property-setting statements we learned about earlier, method `InitializeComponent` now contains event-handler registrations, as follows:

```
C# code

this.MouseDown += new System.Windows.Forms.MouseEventHandler(this.Form1_MouseDown);
this.Closing += new System.ComponentModel.CancelEventHandler(this.Form1_Closing);
this.KeyPress += new System.Windows.Forms.KeyPressEventHandler(this.Form1_KeyPress);
```

Visual Basic uses a different approach, as we'll see later.

Empty skeletons of these event handling methods have also been generated. Code then has to be added manually to say what to do when such an event occurs. Here's `Form1_Closing`:

```
C# code

private void Form1_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
        MessageBox.Show("Handling event for the form\n"
                + "having the title '" + ((Form1)sender).Text + "'",
                "Form Closing Event",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information
        );
}

VB code

Private Sub Form1_Closing(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
        MessageBox.Show("Handling event for the form" + Chr(10) _
                + "having the title '" + CType(sender, Form1).Text + "'", _
                "Form Closing Event", _
                MessageBoxButtons.OK, _
                MessageBoxIcon.Information _
        )
End Sub
```

The first parameter, `sender`, lets us find out which control caused the event, while the second parameter, `e`, is an instance of the event type. Now while we know for sure without checking (since there is only one control, the form itself) that the sender was an instance of `Form1`, we are given an instance of type `System.Object`, and that type does *not* have a property `Text`. So we need to convert that value to type `Form1` first.

Note the `Handles MyBase.Closing` clause in the last handler above. In Visual Basic this sort of clause it used to register a handler rather than having it be done in the method `InitializeComponent` as is done in the other languages.

Class `MessageBox` has a number of overloaded versions of method `Show`. The one used here takes four arguments: a message text string, a MessageBox title string, an indicator as to which buttons should appear in the MessageBox, and the icon that should be displayed next to the message.

Here's `Form1_MouseDown`:

```
C# code

private void Form1_MouseDown(object sender, System.Windows.Forms.MouseEventArgs e)
{
        MessageBox.Show("'MouseDown' detected\n"
                + "Button pressed: " + e.Button
        );
}

VB code

Private Sub Form1_MouseDown(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseDown
        MessageBox.Show("'MouseDown' detected" + Chr(10) + "Button pressed: " _
                + e.Button.ToString())
End Sub
```

This version of `Show` takes only one argument, a message text string, in which case, the MessageBox title is empty, an OK button appears by default, and there is no message icon.

Finally, here's `Form1_KeyPress`: