Chapter 2
A Tour of Windows Forms

At this point I am sure you are anxious to create your first Windows program using C# .NET. You will be surprised at how simple it is. A Windows program created with C# .NET is known as a Windows Form. This is a somewhat misleading term since your program is by no means limited to any type of data entry type of form. You can think of a form as a window. Virtually all Windows programs have one or more forms. It is actually possible to create a Windows program that does not have a form, but that would be unusual. Even programs that appear to be just a dialog are actually Windows Form's programs. A dialog is just a special version of a form.

In this chapter we will take a quick tour of the capabilities of Windows Forms. I want to give you the overall picture quickly so you can have a better grasp of what would be involved in a complete application. Unfortunately many books don’t get to important topics like menus for a couple of hundred pages. Rather, they drill down into all the details on each aspect of Windows Forms programming. We will, of course, cover many topics in depth at the appropriate time. After this whirlwind tour we will visit each aspect of what we cover here in greater detail to start writing more complex programs. However, I won't cover the more advanced details until later chapters. I have found this multi-level approach to be the best way to learn new material. So let’s get started.

If you have never written a program that used a graphical user interface, GUI, there are significant differences. The typical console (command line) application does not involve such features as graphics, menus, buttons, mouse actions, etc. Each of these capabilities requires different programming techniques. One important characteristic of a Windows program is that it is event driven. This means that you don’t control the flow of the program as you do in a console application. Rather, the user of the program determines what action occurs next. The user might click the mouse on a menu item, type some text, or perform any one of the many operations you allow. Your program must be prepared to act on any one of these actions. To make it easy to write programs like this, the Windows operating system generates events that are passed to your program in response to the user’s actions. You provide event handlers in your program to process these events. Stay tuned and you will quickly see what I mean.

Using the New Project Wizard

Let’s start by using the new project wizard to generate a skeleton Windows Form. Select File|New|Project and then Windows Application. Once again you can name your project anything you want and locate it in other than
the default folder. Figure 2-1 shows what your initial application development environment should look like. Your particular layout may be slightly different as Visual Studio’s layout can be customized to your liking.

![Visual Studio window showing Form1.cs and Solution Explorer]

**Figure 2-1**

On the left is what your application’s main form will look like. You can resize this form by dragging the handles along the edges. Using the form’s properties you can change other aspects of the look and feel of the form. I will show you how to do this later. Right now let’s take a preliminary look at the code that the wizard generated. To do this you can right click anywhere in the form and select *View Code*. Here is the code that was generated.

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
```
using System.Windows.Forms;

namespace WindowsApplication2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}

The first thing you will notice is that there are more namespaces included by default. One of the most important is System.Windows.Forms. This namespace contains many of the classes central to Windows Forms programming. The line

public partial class Form1 : Form

defines the class that represents your form. There are two important details to explain. The keyword partial is a new feature of C# 2.0. It allows a class to be broken into two or more disk files. This allows the Visual Studio 2005 forms wizard to place code that the programmer normally doesn’t need to modify in a separate file. We will take a look at the remainder of our class further down the line. Right now it’s better to ignore it. The second important point is that our class, Form1, is derived from the class Form. For programmers not familiar with the concept of inheritance, it means that your class inherits all the feature of the class Form that is in the System.Windows.Forms namespace. This is one of the most important classes since it contains many important methods and properties to manage your form. In other words, we are starting with a generic form and then customizing it into our specific application’s needs.

The method Form1 is the constructor. A constructor is a special method that is used to initialize a class. It always has the same name as the class and never has a return type. Constructors may be overloaded. The method InitializeComponent is being called to set up the controls that you place on your form using the designer. You will learn all about the designer shortly. The designer allows the rapid development of applications by letting you literally drag controls to the form. Typical controls are buttons, check boxes and many more.

The keyword public indicates that the method can be accessed by any other method in your program regardless of what class that method is in. Two other important access keywords are private and protected. These three keywords are similar to the same keywords used in other object oriented languages such as C++.

You can build and execute this application all in one step by selecting Debug|Start Without Debugging or optionally with Debug|Start to run it under the
debugger. As with a console application, if you have any errors the program will not start up and you will have an opportunity to correct your mistakes. If you just want to make sure there are no compiler errors you can select Build|Build Solution. Your application will be compiled but not executed. If you run this application it will look just like it does in the designer pane of Visual Studio. Of course this application doesn't do anything useful but, as you can see, it has the standard features of a normal window including the ability to be minimized, maximized, resized, moved, and closed.

The first thing you might want to do with your new form is to display some output in it. Unfortunately we can't just use the Console class with a Windows Form. A window is a basic element of the Windows family of operating systems. Our form is really a window as far as the operating system is concerned. In order to display something in a window I need to explain the concept of painting. This is what we refer to actions required to place output on the surface of a window. The innermost area of a window is known as the client area. It excludes the borders, title bar, menu (if the window has one), and things like scroll bars. This is the area of the window where we normally provide output. Think of it as the innermost blank area of the form.

**Paint Events**

The Windows operating system does not have any idea what to display in the client area of your window. Unlike a console application we can't display output in our form in the Main method. We need to respond to paint events. Windows sends your program a special message each time it needs to know what to display in the client area of your form. You might ask why we can't just generate the output once and be done with it? The reason is quite simple. Suppose a user minimizes a window to the task bar and later restores it. Windows doesn't remember what is in windows that are not on the screen. Rather, it recreates the entire window so your program needs to provide the content for the client area each time Windows needs to redraw a window. The paint event is the event sent to your program each time windows needs this information. The Windows operating system generates messages. These messages are in turn used to generate the events passed to your application. All the details of this rather complex mechanism are handled by the .NET framework. Paint events are generated for the following actions:

- Restoring a window from the task bar.
- Uncovering a portion of a window that is covered by another window.
- At the request of your program itself.

A paint event is not generated when you merely move a window unless this action results in some portion of the window becoming visible that was previously covered up or off screen.
Adding a paint event is very easy using the powerful features of the Visual Studio IDE. Start by right clicking inside the designer view of your form and then select Properties. This will bring up the properties pane for your form class. It is most likely in the lower right hand corner of your Visual Studio window. Click on the AZ button as well as the button that looks like a lightening bolt. The properties pane should look something like this:

![Properties pane](image)

The next step is to double click on "Paint." Scroll down if this line is not visible. This results in a paint event being added to your program. The following function is added to your form:

```csharp
private void Form1_Paint(object sender, PaintEventArgs e) {
}
```

There is actually a line of code that is added to the designer generated file, but again we will ignore it for the moment. This method will be called every time that Windows needs to have you repaint your form's client area. The next step is to explore the methods that allow us to output text and graphics to the client area.

**The Graphics Class**

The Windows operating system provides the capabilities to output text and graphics to a window through the Graphics Device Interface, or GDI. A newer
version of this interface is known as GDI+. .NET uses GDI+ to allow the use of the more advanced features it provides. The GDI+ interface is encapsulated by the Framework Class Library. The most important class is the Graphics class. There are a whole host of methods we can use. An exhaustive coverage of these methods would fill an entire book. However, once are familiar with the basic concepts you can investigate the .NET documentation to find all the goodies provided by the Graphics class. This class is in the System.Drawing namespace. The System.Drawing namespace is included by default in a Windows Forms project.

In order to use the Graphics class we need to obtain a reference to an instance of this class that is specific to the client area of our form. Fortunately this is extremely simple. The second parameter passed to our paint event handler, `e`, is a reference to a PaintEventArgs class instance. The Graphics property of this class is the reference to the instance of the Graphics class that we need. Some programmers new to C++ may get a bit confused over the use of the same identifier to refer to different things. For example, Graphics is the name of a class as well as a property within the PaintEventArgs class. It is usually easy to determine what the identifier is referring to by its context. Rather than always using `e.Graphics` we can create a new reference, `g`, and initialize it to `e.Graphics` like this:

```csharp
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
}
```

Now we can use `g` to refer to the methods in the Graphics class.

A very important method is `DrawString`. It has six overloads. Let's start with one of the most basic.

```
DrawString(s, font, brush, x, y)
```

`S` is a reference to a `string` and is the text to display. `Font` and `brush` are references to a Font and Brush to use to display the text. `X` and `y` represent the coordinate of the upper left hand corner of the line of text. This coordinate defaults to the upper left hand corner of the client area of your form and is in units of pixels. Positive X is to the right and positive Y is down. This may not be what you expected.
We will discuss other possible coordinate systems we can use later on. The
default coordinate system is often referred to as *device coordinates*. These last
two parameters are of the float data type but it makes no sense to use anything
but integer values. However, you will learn that we can change the units used for
representing a coordinate to something else where fractional numbers have more
meaning. The following code displays a line at coordinate 10, 10. This is ten
pixels to the right and below the upper left hand corner of the client area:

```csharp
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Here is a line of text.", Font,
                 Brushes.Black, 10, 10);
}
```

Figure 2-3 shows what our form looks like when we run the program:
To specify the font I used the Font property of the Form class. Remember, we inherit all the properties, fields, and methods from the base class, Form. The Font property is set to the default font for forms. This font depends on your operating system version and culture settings. Later we will see how to use a font of your choice. The Brushes collection contains a set of brushes in many standard colors for you use if you don’t want to create a brush in a custom color. Here I have selected a standard black brush. Each brush is actually a static property of the Brushes class.

We can display graphics objects using the Graphics class. The following method first displays a cyan colored rectangle and then our line of text.

```csharp
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillRectangle(Brushes.Cyan, 0, 0, 50, 50);
    g.DrawString("Here is a line of text.", Font, Brushes.Black, 10, 10);
}
```

As you can see in Figure 2-4 the text is displayed on top of the rectangle. The order that things are drawn is the order the methods are called. If we reverse the order of these functions we get the result shown in Figure 2-5.
There are many methods in the Graphics class and we could cover them all right now, but you would be bored quite quickly. All you need to keep in mind right now is that the Graphics class contains the methods you use to paint on the client area of your form. Later on we can explore some of the more sophisticated methods in the Graphics class and by then you will be eager to learn about them.
Mouse Clicks

Few Windows applications exist that don’t use the mouse in some way. Of course we use the mouse to click on menu selections, scroll a window, etc. What we will take a look at now is the use of the mouse when you click inside the client area of your form. Handling mouse events is actually one of the easiest things we can do thanks to the capabilities of the Framework Class Library. Just like the paint event there is an event generated when you click any button on your mouse as well as perform other tasks such as moving the mouse, activating the wheel if your mouse has one, and other features found on some more sophisticated mice.

Start by adding an event handler for the MouseClick event. Do this in the same way as you did for the Paint event. The following code is added to your form:

```csharp
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
}
```

The only argument we need to be concerned with is the second, e, which is a reference to an instance of the MouseEventArgs class. The basic information we normally want to know is which mouse button was clicked and exactly where in the client area we clicked. The following method obtains this information:

```csharp
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    int x, y;
    x = e.X;
    y = e.Y;
    if (e.Button == MouseButtons.Left)
    {
        //do something for left button
    }
}
```

The integers x and y are set to the coordinate of the mouse click in pixels. This position is relative to the upper left hand corner of the client area. As you can see, we can check if we clicked the left mouse button. We could also check the middle or right mouse button as well. The Button property tells us which button was clicked and it evaluates to the enumerated data type MouseButtons.

So now we can perform some operation if we click the left mouse button. We know where we clicked and can use that information if we wish. But what can we do? Remember, we are in the event handler for the mouse click event. We can’t just call a method in the Graphics class since we don’t have a reference to an instance of this class we can use. Actually there is a way to obtain such an instance and we could indeed paint to our form. We could, for example, output a
graphic at the mouse position, or perhaps some text. Let's say we did this. Is there a problem? There is a problem not obvious to someone new to programming for the Windows operating system. If we were to minimize and then restore our form a paint event would be generated. Since nothing is remembered, whatever we painted in the mouse event handler would have been lost. Unless we somehow remember what we had previously painted we can't possibly restore the contents of our form.

The accepted solution to this problem is to always be able to repaint the client area of your form. This means we need to save whatever information is needed to do this. Suppose we wanted to display a small black circle where we click the mouse. All we would need to do is to store the coordinate where we last clicked and cause our form to repaint itself. The following code does just this:

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MouseClick
{
    public partial class Form1 : Form
    {
        private int x;   //mouse x coord
        private int y;   //mouse y coord

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Left)
            {
                x = e.X;
                y = e.Y;
                this.Invalidate();
            }
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            const int WIDTH = 20;
            const int HEIGHT = 20;

            Graphics g = e.Graphics;
            g.FillEllipse(Brushes.Black, x-WIDTH/2, y-WIDTH/2, WIDTH, HEIGHT);
        }
    }
}
```
The first change I made was to move the x and y variables outside the event handler and into the class as fields. This keeps them around after we return. As the action to perform when the left button is clicked I call the Invalidate method of the Form class. This is an inherited method and causes the form's client area to be repainted. The this keyword is not strictly required, but is makes it clear that we are invoking a method that is inherited by, or part of our class. We can often use the this keyword to our advantage to eliminate an ambiguous reference. Our paint event handler is very simple. The call to FillEllipse draws our black dot centered on our mouse click. The second and third parameters position the upper left hand corner of the bounding rectangle for the ellipse. We need to offset this from our mouse click by one half the width and height in the negative direction. The last two parameters specify the width and height of our ellipse. In this case it is a circle since the width and height are equal. Figure 2-6 shows what our program looks like when we execute it.

![Form1](image)

What if we wanted to display more than one circle? We could do that if we used a data structure such as an array to store a list of coordinates. When we paint we can then iterate through the list and display each point. Let's, do that. This will introduce us to some powerful features of the FCL. Here is a program that accomplishes this task:
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Collections;

namespace MouseClick
{
    public partial class Form1 : Form
    {
        private ArrayList coordinates = new ArrayList();

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_MouseClick(object sender, MouseEventArgs e)
        {
            if (e.Button == MouseButtons.Left)
            {
                Point p = new Point(e.X, e.Y);
                this.coordinates.Add(p);
                this.Invalidate();
            }
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            const int WIDTH = 20;
            const int HEIGHT = 20;

            Graphics g = e.Graphics;
            foreach (Point p in this.coordinates)
            {
                g.FillEllipse(Brushes.Black,
                               p.X - WIDTH/2, p.Y - HEIGHT/2, WIDTH, HEIGHT);
            }
        }
    }
}

To build the array of coordinates I used the ArrayList class of the System.Collections namespace. Be sure to include this namespace. It is not included by default. The advantage of the ArrayList class is that it automatically grows as needed and can be an array of any data type. However, it must be an array of a single data type. We could store first the x and then the y value from each coordinate since they are both integers. We would then have to extract each coordinate in pairs. This would be fine except that it really doesn’t embrace
the tenants of object oriented programming. Fortunately there is a class just waiting for us, the \textit{Point} class. We will see a lot of this class in the chapters to come. Our mouse event handler merely constructs a new \textit{Point} object and adds it to our \textit{ArrayList} object named \textit{coordinate}. We create the \textit{coordinate} object using the \textit{new} operator. If we don't do this then we will get compiler error. Our paint event handler demonstrates the power of the \textit{foreach} statement of C#. \textit{Foreach} will loop through all the elements that are currently in our list of coordinates. Figure 2-7 shows the output if you execute this program and make a few mouse clicks.

![Form1](image)

\textbf{Figure 2-7}

Our last addition to this program will be a way of clearing all the points we have remembered. In other words this will clear the client area. This mouse event handler does the trick:

```csharp
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
    {
        Point p = new Point(e.X, e.Y);
        this.coordinates.Add(p);
        this.Invalidate();
    }
    if (e.Button == MouseButtons.Right)
    {
        this.coordinates.Clear();
        this.Invalidate();
    }
}
```
When we click on the right mouse button we call the `Clear` method of the `ArrayList` class. This removes all the elements. We need to repaint the form for this change to be reflected and we do this as we did with the left mouse click by a call to `Invalidate`.

**Controls**

Controls are special types of windows. Many controls are actually implemented as an intrinsic part of the Windows operating system for greater efficiency, but there are some controls that you will find only in the .NET FCL. All controls are derived from the `Control` class. The `Control` class implements very basic functionality required by classes that display information to the user. It handles user input through the keyboard and pointing devices. It handles message routing and security. It defines the bounds of a control (its position and size), although it does not implement painting. We don’t use an instance of the `Control` class directly, but rather use classes derived from `Control`. Some of the controls you use every day include:

- Buttons
- Check boxes
- Edit controls
- Radio buttons
- List controls

You can place controls directly in the client area of your form just like you draw text or graphics. There is one big difference. Normally you don’t ever need to become involved with painting the control itself. Controls are actually child windows to your form’s window. The Windows operating system handles making paint requests for these child windows when they need updating. Even better, you don’t need to write the `Paint` event handler for these controls. Painting is implemented for you. So how do we include a control in the client area of our form? Let’s say we want to place a nice button with the label “Clear” in the client area of our last example. We ultimately want this button to clear the client area just like the right mouse button does.

Although we could type in all the code required to implement our button, Visual Studio 2005 provides a much easier method in keeping with rapid application development. The `Toolbox` is a docking window that contains a tree structured list of controls you can use with your application. You can open the `Toolbox` if it is not visible using View|Toolbox from the menu. There is also a button on the Standard toolbar that does the same thing as well. If you expand the `Common Controls` branch you will see a list of the most frequently used
controls. As you can see in Figure 2-8 the button control is right at the top. There are two ways we can place our control in the client area of our form. You can drag the button control from the Toolbox to our form in the Designer window. This results in a standard size button. Alternatively you can first click on the button selection in the Toolbox and then click in the client area of your form. You then need to hold down the left mouse button and drag a box to the desired size you want you button. This technique works with many other controls as well. Figure 2-9 shows how things should look if you drag the button to your form.

![Toolbox with various controls](image-url)

*Figure 2-8*
All we need to do now is to change the label on the button and implement an event handling for the *click* event for the button. First right click on the button in the designer and bring up the properties window if it is not already displayed. You can get an alphabetical listing of all the button's properties by clicking on the A-Z button on the properties window's toolbar. Find the *Text* property and change the text to "Clear." You can add a *Click* event handler just like you added the *Paint* event handler by finding the event and double clicking. Remember to first open the events drop down list by clicking on the lightening bolt button in the properties window toolbar. You will see an empty method for our *click* event handler. All we need to do is add the two lines of code that we used with our right mouse button handler to duplicate the action. Now we have two ways to clear the client area of our small circles. Here is the event handler code:

```csharp
private void button1_Click(object sender, EventArgs e)
{
    this.coordinates.Clear();
    this.Invalidate();
}
```

**The Designer and Hidden Code**

The designer automatically generates code when you use any of its features such as dragging a control to your form. Prior to Visual Studio 2005 this code was in the same file as your form, e.g., *Form1.cs*. With Visual Studio 2005 the new *partial* keyword allows the code generated by the designer to be
namespace MouseClick
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(95, 30);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.TabIndex = 0;
            this.button1.Text = "Clear";
            this.button1.Click += new System.EventHandler(this.button1_Click);
            //
            // Form1
            //
            this.ClientSize = new System.Drawing.Size(612, 453);
            this.Name = "Form1";
            this.Text = "Form1";
            //
            // Form1.cs
            //
            this.Text = "Form1 - Mouse Click Program";
            //
            // Form1_Load
            //
            this.button1.Click += new System.EventHandler(this.button1_Click);
            //
            // Form1_Load
            //
            this.ResumeLayout(false);
        }

        #endregion
    }
}

If you open this file you may not see all the code shown above. A feature of Visual Studio 2005 and C# is the #region and #endregion directives. These directives don’t affect the program in any way. They serve only to bracket sections of your code that can be collapsed in your edit window. You can expand this code by clicking on the small plus sign at the left margin.

Recall that we added event handlers for the Paint event and the MouseClick event. The methods to process these events are placed in the main file for your form and not in the hidden code file generated by the designer. However, these methods must be connected to the actual event. The following statements handle this:

```csharp
this.Paint += new System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
this.MouseClick += new System.Windows.Forms.MouseEventHandler(this.Form1_MouseClick);
```

A complete understanding of how this all works involves learning about delegates and events. We will cover this in chapter <ref>. Adding the button involves more code. The button is actually a class in the .NET FCL. In order to use this class we need to create an instance of this class. A field is added to our class to hold a reference to the instance we will create. Here is the line that accomplishes this:

```csharp
private System.Windows.Forms.Button button1;
```
A number of statements are provided to create the instance of the Button class and to initialize some of its properties.

```csharp
this.button1 = new System.Windows.Forms.Button();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(95, 30);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "Clear";
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Some of these statements are straightforward and others are not. Right now you don’t need to understand the details. It is sufficient to just understand how the designer works. While it is possible to edit this code directly it is strongly discouraged. For example, the size of the button is shown by this statement:

```csharp
this.button1.Size = new System.Drawing.Size(75, 23);
```

If we wanted to resize the button to (100,50) we could edit this statement and our button would be displayed in its new size. A better way to do this is to just merely drag the sizing handles of the button in the designer. If you want a more precise size down to the pixel you can use the properties window to type in the width and height. The code in the hidden class will be adjusted automatically. Impressive, isn’t it?

Many programmers feel that in order to be a good programmer you must type everything in yourself and understand each and every line of code, method call, etc. I used to have this philosophy. There are two drawbacks to this approach. First is the effort involved in typing the same things over and over. The second is that application frameworks such as .NET are becoming more and more complicated and it is almost impossible to spend the time to learn every detail. On the other hand, I feel that every programmer needs to know enough of the details to be able to drill down, if needed, and analyze what is going on. I have seen many programmers run into brick walls because they do not understand enough of the inner working of their programming tools and the application frameworks they are using. I will strive for a balance as we move through the .NET world.

**Adding a Menu**

Adding a menu to our form is very simple thanks to the designer. The .NET Framework 2.0 has a new menu class, MenuStrip. This replaces the older
MainMenu class found in earlier FCL versions. The toolbox has a branch calledMenus & Toolbars where you can find this control. Just drag it to your form in design view. Figure 2-10 Shows how things should look at this point.

![Figure 2-10](image)

Next we need to add the items that form our top level menu selections. For this example we will only have a File menu item. Even though we don’t have any file operations yet I will give it this name since we are used to seeing a File item in the leftmost position. Click on the text “Type Here” and then type ”&File.” The & character marks the following character for use in keyboard menu selection. Entering ALT-f will then open the file dropdown when our program is executed. Now add ”&Clear” to the topmost position in the dropdown. Figure 2-11 illustrates our final menu.
All we need to do now is to set up an event handler for our Clear menu item. We can do this in exactly the same way we did it for our Clear button or our mouse click using the properties window for the Clear item. However, if we did this we would have to duplicate the statements we used with the Clear button. There is a trick we can use in the case where two different actions would result in the same code being executed. Since the Clear button does what we want we can invoke the event handler for the Clear button when we click on the Clear menu item. In the event list for the menu item you can click on a dropdown for the Click event and suitable event handlers will be listed. In the case select button1_click as shown in Figure 2-12.
Figure 2-13 shows our running application with the *File* menu expanded. If you click on *Clear*, any displayed dots are removed. Try it.
Summary

In this chapter we took a rapid tour of some of the basic capabilities of creating a window form. At this point you should understand the following concepts:

1. Creating a Windows Forms application using the wizard.
2. Adding a Paint event to allow text and graphics to be displayed in the client area of the form.
3. Drawing text and simple graphics.
4. Handling the MouseClick event.
5. The concept of controls.
6. Adding a Button control and its Click event handler to the form.
8. Adding a menu and processing MenuItem events.

Even with this relatively small amount of knowledge I am sure you can dream up some simple applications using what you have learned. We have lots more to cover, of course, and as I mentioned earlier my approach is to cover each area of Windows Forms programming in greater depth, but not exhaustively, so you can acquire more in depth knowledge as quickly as possible. A second visit will be made to some of the more advanced aspects of each topic. The intent is that you can read what you want when you need it.

Properties – discuss someplace
Need to set up a standard for using italics etc. for names. e.g., first time or all times