

Handling Events

In this chapter you will:

- ⦿ Learn about event handling
- ⦿ Learn about delegates
- ⦿ Declare your own events and handlers and use the built-in `EventHandler`
- ⦿ Handle `Control` component events
- ⦿ Handle mouse and keyboard events
- ⦿ Manage multiple `Controls`
- ⦿ Learn how to continue your exploration of `Controls` and events

Throughout this book, you have learned how to create interactive GUI programs in which a user can manipulate a variety of `Controls`. You have worked with several `Controls` that respond to a user-initiated event, such as a mouse click, and you have provided actions for `Control` default events. In this chapter, you will expand your understanding of the event-handling process. You will learn more about the object that triggers an event and the object that captures and responds to that event. You also will learn about delegates—objects that act as intermediaries in transferring messages from senders to receivers. You will create delegates and manage interactive events. You will learn to manage multiple events for a single `Control` and to manage multiple `Controls` for a project.

Event Handling

In *C#*, an **event** is a reaction to an occurrence in a program; an event transpires when something interesting happens to an object. When you create a class, you decide exactly what is considered “interesting.” For example, when you create a `Form`, you might decide to respond to a user clicking a `Button` but ignore a user who clicks a `Label`—clicking the `Label` is not “interesting” to the `Form`.

A program uses an event to notify a client when something happens to an object. Events are used frequently in GUI programs—for example, a program is notified when the user clicks a `Button` or chooses an option from a `ListBox`. In addition, you can use events with ordinary classes that do not represent GUI controls. When an object’s client might want to know about any changes that occur in the object, events enable the object to signal the client.



The actions that occur during the execution of a program occur **at runtime**. The expression *at runtime* is used to distinguish execution-time activities from those that occur during development time and compile time. Events are runtime occurrences.

You have learned that when a user interacts with a GUI object, an event is generated that causes the program to perform a task. GUI programs are **event driven**—an event such as a button click “drives” the program to perform a task. Programmers also say that an action like a button click **raises an event**, **fires an event**, or **triggers an event**.

For example, Figure 13-1 shows a `Form` that contains a `Label` and a `Button`. The following changes are the only ones that have been made to the default `Form` in the IDE:

- The `Size` property of the `Form` has been adjusted to 500, 120.
- A `Label` has been dragged onto the `Form`, its `Name` property has been set to `helloLabel`, its `Text` property has been set to *Hello*, and its `Font` has been increased to 9.
- A `Button` has been dragged onto the `Form`, its `Name` property has been set to `changeButton`, and its `Text` property has been set to *Change Label*.

- When you double-click the button on the form, Visual Studio generates the following empty method in the program code:

```
private void changeButton_Click(object sender, EventArgs e)
{
}
```

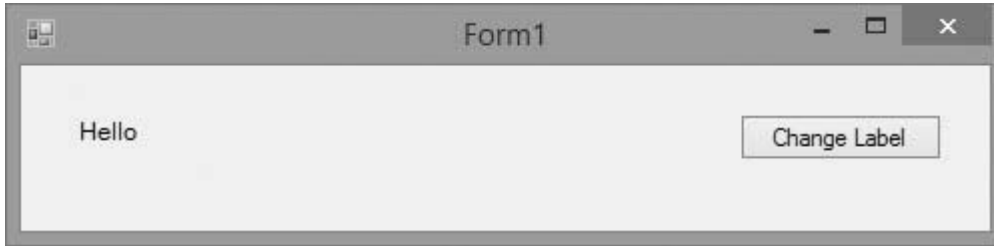


Figure 13-1 A Form with a Label and a Button

A method that performs a task in response to an event is an **event handler**. The `changeButton_Click()` method is an event handler. Although it is legal to create an event handler using any legal identifier, conventionally, event handlers are named using the identifier of the `Control` (in this case, `changeButton`), an underscore, and the name of the event type (in this case, `Click`).

Suppose that when a user clicks the button, you want the text on the label to change from *Hello* to *Goodbye*. You can write the following code for the event handler:

```
private void changeButton_Click(object sender, EventArgs e)
{
    helloLabel.Text = "Goodbye";
}
```

Then, when you run the application and click the button, the output appears as shown in Figure 13-2.

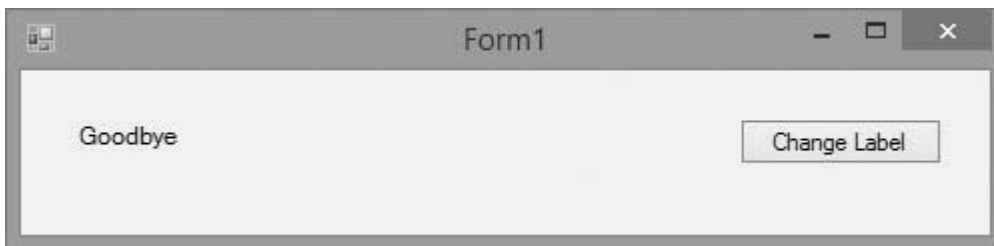


Figure 13-2 Output of the EventDemo application after the user clicks the button

The event-handler method is also known as an **event receiver**. The control that generates an event is an **event sender**. The first parameter in the list for the event receiver method is an object named **sender**; it is a reference to the object that generated the event. For example, if you code the event handler to display **sender**'s information as follows, the output appears as in Figure 13-3.

```
private void changeButton_Click(object sender, EventArgs e)
{
    helloLabel1.Text = sender.ToString();
}
```

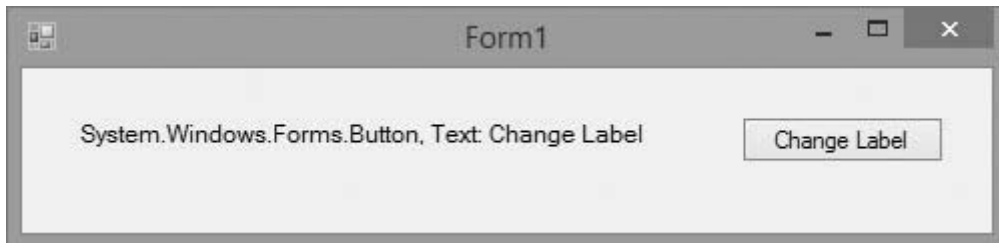


Figure 13-3 The EventDemo application modified to display sender information

The label in Figure 13-3 shows that the sender of the event is an instance of `System.Windows.Forms.Button`, whose `Text` property is `Change Label`.

The second parameter in the event-handler parameter list is a reference to an event arguments object of type `EventArgs`; in this method, the `EventArgs` argument is named `e`. `EventArgs` is a C# class designed for holding event information. If you change the code in the event handler to display the `EventArgs.ToString()` value as in the following and then run the program and click the button, you see the output in Figure 13-4.

```
private void changeButton_Click(object sender, EventArgs e)
{
    helloLabel1.Text = e.ToString();
}
```

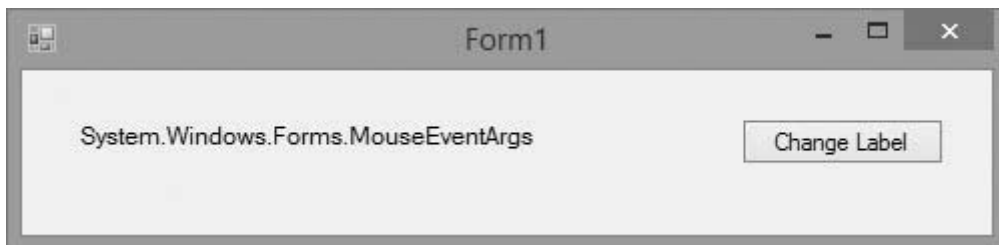


Figure 13-4 The EventDemo application modified to display EventArgs information

In Figure 13-4, you can see that the `e` object is a `MouseEventArgs` object. That makes sense, because the user used the mouse to click the `Button`. (If the user has a touch screen, the user can also touch the `Button` to generate the event.)

When you open the `Designer.cs` file in the IDE, you can examine all the code generated for the application that creates the `Form` shown in Figure 13-4. Expanding the code generated by the Windows Form Designer lets you see comments as well as statements that set the `Controls`' properties. For example, the code generated for `changeButton` appears in Figure 13-5. You can recognize that such features as the button's `Location`, `Name`, and `Size` have been set.

```
//
// changeButton
//
this.changeButton.Location = new System.Drawing.Point(359, 24);
this.changeButton.Name = "changeButton";
this.changeButton.Size = new System.Drawing.Size(101, 23);
this.changeButton.TabIndex = 1;
this.changeButton.Text = "Change Label";
this.changeButton.UseVisualStyleBackColor = true;
this.changeButton.Click += new
    System.EventHandler(this.changeButton_Click);
```

Figure 13-5 Code involving `changeButton` generated by Visual Studio



The code generated in Design mode in the IDE is not meant to be altered by typing. You should modify `Control` properties through the Properties window in the IDE, not by typing in the `Designer.cs` file.

All the code in Figure 13-5 was created by Visual Studio when the programmer clicked `changeButton` in the IDE. The most unusual statement in the section of `changeButton` code is shaded in Figure 13-5. This statement concerns a **click event**, which is an action fired when a user clicks a button during program execution. The shaded statement is necessary because `changeButton` does not automatically “know” what method will handle its events—`C#` and all other `.NET` languages allow you to choose your own names for event-handling methods for events generated by GUI objects. In other words, the event-handling method is not required to be named `changeButton_Click()`. You *could* create your program so that when the user clicks the button, the event-handling method is named `calculatePayroll()`, `changeLabel()`, or any other identifier for a method you could then write. Of course, you do not want to make such a change; using the name `changeButton_Click()` for the method that executes when `changeButton` is clicked is the clearest approach.

Connecting an event to its resulting actions is called **event wiring**. The event wiring for the `changeButton_Click()` method is accomplished in the shaded statement in Figure 13-5. The statement indicates that, for this program, the method named `changeButton_Click()` is the receiver for `changeButton`'s `Click` event; the method executes when a user clicks the button and an event is fired. Programmers say the statement creates a *delegate* or, more specifically,

a *composed delegate*. The delegate's type is `EventHandler`, and it takes a reference to the `changeButton_Click()` method. You will learn about delegates in the next two sections of this chapter, and you will also learn why the `+=` operator is used in the statement.

TWO TRUTHS & A LIE

Event Handling

1. An action such as a key press or button click raises an event.
2. A method that performs a task in response to an event is an event handler.
3. The control that generates an event is an event receiver.

The false statement is #3. The control that generates an event is an event sender.

Understanding Delegates

A **delegate** is an object that contains a reference to a method; object-oriented programmers would say that a delegate *encapsulates a method*. In government, a delegate is a representative that you authorize to make choices for you. For example, states send delegates to presidential nominating conventions. When human delegates arrive at a convention, they are free to make last-minute choices based on current conditions. Similarly, *C#* delegates provide a way for a program to take alternative courses that are not determined until runtime. When you write a method, you don't always know which actions will occur at runtime, so you give your delegates authority to run the correct methods.



In Chapter 1, you learned that encapsulation is a basic feature of object-oriented programming. Recall that encapsulation is the technique of packaging an object's attributes and methods into a cohesive unit that can then be used as an undivided entity.

After you have instantiated a *C#* delegate, you can pass this object to a method, which then can call the method referenced within the delegate. In other words, a delegate provides a way to pass a reference to a method as an argument to another method. In other words, although you can't pass a method to a method, you *can* pass an object that is a reference to a method. For example, if `del` is a delegate that contains a reference to the method `M1()`, you can pass `del` to a new method named `MyMethod()`. Alternatively, you could create a delegate named `del` that contains a reference to a method named `M2()` and then pass this version to `MyMethod()`. When you write `MyMethod()`, you don't have to know whether it will call `M1()` or `M2()`; you only need to know that it will call whatever method is referenced within `del`. Perhaps the decision about

which method will execute will be made as the result of user input the date of execution, or some other factor.



A C# delegate is similar to a function pointer in C++. A function pointer is a variable that holds a method's memory address. In the C++ programming language, you pass a method's address to another method using a pointer variable. Java does not allow function pointers because they are dangerous—if the program alters the address, you might inadvertently execute the wrong method. C# provides a compromise between the dangers of C++ pointers and the Java ban on passing functions. Delegates allow flexible method calls, but they remain secure because you cannot alter the method addresses.

You declare a delegate using the keyword `delegate`, followed by an ordinary method declaration that includes a return type, method name, and argument list. For example, by entering the following statement, you can declare a delegate named `GreetingDelegate()`, which accepts a `string` argument and returns nothing:

```
delegate void GreetingDelegate(string s);
```

Any delegate can encapsulate any method that has the same return type and parameter list as the delegate. So `GreetingDelegate` can encapsulate any method as long as it has a `void` return type and a single `string` parameter. If you declare a delegate and then write a method with the same return type and parameter list, you can assign an instance of the delegate to represent it. For example, the following `Hello()` method is a `void` method that takes a `string` parameter:

```
public static void Hello(string s)
{
    WriteLine("Hello, {0}!", s);
}
```

Because the `Hello()` method matches the `GreetingDelegate` definition, you can assign a reference to the `Hello()` method to a new instance of `GreetingDelegate`, as follows:

```
GreetingDelegate myDel = new GreetingDelegate(Hello);
```

Once the reference to the `Hello()` method is encapsulated in the delegate `myDel`, each of the following statements will result in the same output: "Hello, Kim!"

```
Hello("Kim");
myDel("Kim");
```

In this example, the ability to use the delegate `myDel` does not seem to provide any benefits over using a regular method call to `Hello()`. If you have a program in which you pass the delegate to a method, however, the method becomes more flexible; you gain the ability to send a reference to an appropriate method you want to execute at the time.

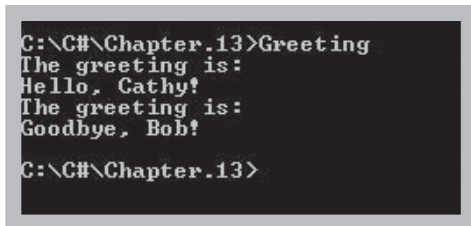
For example, Figure 13-6 shows a `Greeting` class that contains `Hello()` and `Goodbye()` methods. The `Main()` method declares two delegates named `firstDel` and `secondDel`. One is instantiated using the `Hello()` method, and the other is instantiated using the `Goodbye()` method. When the `Main()` method calls `GreetMethod()` two times, it passes a different method and string each time. Figure 13-7 shows the output.

```

using static System.Console;
delegate void GreetingDelegate(string s);
class Greeting
{
    public static void Hello(string s)
    {
        WriteLine("Hello, {0}!", s);
    }
    public static void Goodbye(string s)
    {
        WriteLine("Goodbye, {0}!", s);
    }
    static void Main()
    {
        GreetingDelegate firstDel, secondDel;
        firstDel = new GreetingDelegate(Hello);
        secondDel = new GreetingDelegate(Goodbye);
        GreetMethod(firstDel, "Cathy");
        GreetMethod(secondDel, "Bob");
    }
    public static void GreetMethod(GreetingDelegate gd, string name)
    {
        WriteLine("The greeting is:");
        gd(name);
    }
}

```

Figure 13-6 The Greeting program



```

C:\C#\Chapter.13>Greeting
The greeting is:
Hello, Cathy!
The greeting is:
Goodbye, Bob!

C:\C#\Chapter.13>

```

Figure 13-7 Output of the Greeting program



Delegates are useful in that a method reference can be passed to another method, so the receiving method can be customized. As with many other features of C# (and all other programming languages), you can use other techniques to accomplish the same result. For example, you certainly can produce simple output such as *Hello, Cathy* using other techniques. However, you should understand at least a little about delegates because they are used in the code that responds to events in C#'s GUI environment.

Creating Composed Delegates

You can assign one delegate to another using the = operator. You also can use the + and += operators to combine delegates into a **composed delegate** that calls the delegates from which it is built. As an example, assume that you declare three delegates named `de11`, `de12`, and `de13`, and that you assign a reference to the method `M1()` to `de11` and a reference to method `M2()` to `de12`. When the statement `de13 = de11 + de12;` executes, `de13` becomes a delegate that executes both `M1()` and `M2()`, in that order. Only delegates with the same parameter list can be composed, and the delegates used must have a `void` return type. Additionally, you can use the - and -= operators to remove a delegate from a composed delegate. A composed delegate is a collection of delegates. The += and -= operators add and remove items from the collection.

Figure 13-8 shows a program that contains a composed delegate. This program contains only two changes from the Greeting program in Figure 13-6: the class name (`Greeting2`) and the shaded statement that creates the composed delegate. The delegate `firstDel` now executes two methods, `Hello()` and `Goodbye()`, whereas `secondDel` still executes only `Goodbye()`. Figure 13-9 shows the output; *Cathy* is used with two methods, but *Bob* is used with only one.

```
using static System.Console;
delegate void GreetingDelegate(string s);
class Greeting2
{
    public static void Hello(string s)
    {
        WriteLine("Hello, {0}!", s);
    }
    public static void Goodbye(string s)
    {
        WriteLine("Goodbye, {0}!", s);
    }
    static void Main()
    {
        GreetingDelegate firstDel, secondDel;
        firstDel = new GreetingDelegate(Hello);
        secondDel = new GreetingDelegate(Goodbye);
        firstDel += secondDel;
        GreetMethod(firstDel, "Cathy");
        GreetMethod(secondDel, "Bob");
    }
    public static void GreetMethod
    (GreetingDelegate gd, string name)
    {
        WriteLine("The greeting is:");
        gd(name);
    }
}
```

Figure 13-8 The Greeting2 program

```
C:\C#\Chapter.13>Greeting2
The greeting is:
Hello, Cathy!
Goodbye, Cathy!
The greeting is:
Goodbye, Bob!
C:\C#\Chapter.13>
```

Figure 13-9 Output of the Greeting2 program



Watch the video *Event Handling*.

TWO TRUTHS & A LIE

Understanding Delegates

1. A delegate is an object that contains a reference to a method.
2. Once you have created a delegate, it can encapsulate any method with the same identifier as the delegate.
3. A composed delegate can be created using the += operator; it calls the delegates from which it is built.

The false statement is #2. Once you have created a delegate, it can encapsulate any method with the same return type and parameter list as the delegate.



You Do It

Creating Delegates

To demonstrate how delegates work, you create two delegate instances in the next steps and assign different method references to them.

1. Open a new console project named **DiscountDelegateDemo**. Type the necessary `using` statements, and then create a delegate that encapsulates a void method that accepts a `double` argument:

```
using System;  
using static System.Console;  
delegate void DiscountDelegate(ref double saleAmount);
```

(continues)

(continued)

2. Begin creating a `Discount` class that contains a `StandardDiscount()` method. The method accepts a reference parameter that represents an amount of a sale. If the sale amount is at least \$1,000, a discount of 5 percent is calculated and subtracted from the sale amount. If the sale amount is not at least \$1,000, nothing is subtracted.

```
class DiscountDelegateDemo
{
    public static void StandardDiscount(ref double saleAmount)
    {
        const double DISCOUNT_RATE = 0.05;
        const double CUTOFF = 1000.00;
        double discount;
        if(saleAmount >= CUTOFF)
            discount = saleAmount * DISCOUNT_RATE;
        else
            discount = 0;
        saleAmount -= discount;
    }
}
```

3. Add a `PreferredDiscount()` method. The method also accepts a reference parameter that represents the amount of a sale and calculates a discount of 10 percent on every sale.

```
public static void PreferredDiscount(ref double saleAmount)
{
    const double SPECIAL_DISCOUNT = 0.10;
    double discount = saleAmount * SPECIAL_DISCOUNT;
    saleAmount -= discount;
}
}
```

4. Start a `Main()` method that declares variables whose values (a sale amount and a code) will be supplied by the user. Declare two `DiscountDelegate` objects named `firstDel` and `secondDel`. Assign a reference to the `StandardDiscount()` method to one `DiscountDelegate` object and a reference to the `PreferredDiscount()` method to the other `DiscountDelegate` object.

```
static void Main()
{
    double saleAmount;
    char code;
    DiscountDelegate firstDel, secondDel;
    firstDel = new DiscountDelegate(StandardDiscount);
    secondDel = new DiscountDelegate(PreferredDiscount);
}
```

(continues)

(continued)

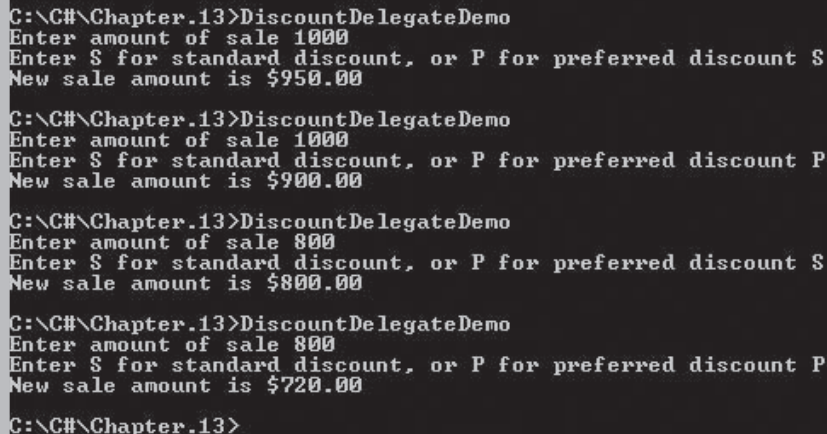
- Continue the `Main()` method with prompts to the user to enter a sale amount and a code indicating whether the standard or preferred discount should apply. Then, depending on the code, use the appropriate delegate to calculate the correct new value for `saleAmount`. Display the value and add closing curly braces for the `Main()` method and the class.

```

Write("Enter amount of sale ");
saleAmount = Convert.ToDouble(ReadLine());
Write("Enter S for standard discount,"
      + "or P for preferred discount ");
code = Convert.ToChar(ReadLine());
if(code == 'S')
    firstDel(ref saleAmount);
else
    secondDel(ref saleAmount);
WriteLine("New sale amount is {0}",
          saleAmount.ToString("C2"));
    }
}

```

- Save the file, and then compile and execute it. Figure 13-10 shows the results when the program is executed several times.



```

C:\C#\Chapter.13>DiscountDelegateDemo
Enter amount of sale 1000
Enter S for standard discount, or P for preferred discount S
New sale amount is $950.00

C:\C#\Chapter.13>DiscountDelegateDemo
Enter amount of sale 1000
Enter S for standard discount, or P for preferred discount P
New sale amount is $900.00

C:\C#\Chapter.13>DiscountDelegateDemo
Enter amount of sale 800
Enter S for standard discount, or P for preferred discount S
New sale amount is $800.00

C:\C#\Chapter.13>DiscountDelegateDemo
Enter amount of sale 800
Enter S for standard discount, or P for preferred discount P
New sale amount is $720.00

C:\C#\Chapter.13>

```

Figure 13-10 Sample executions of the `DiscountDelegateDemo` program

(continues)

(continued)

Creating a Composed Delegate

When you compose delegates, you can invoke multiple method calls using a single statement. In the next steps, you create a composed delegate to demonstrate how composition works.

1. Open the **DiscountDelegateDemo** file, and immediately save it as **DiscountDelegateDemo2**.
2. Within the `Main()` method, add a third `DiscountDelegate` object to the statement that declares the two existing versions, as follows:
DiscountDelegate firstDel, secondDel, thirdDel;
3. After the statements that assign values to the existing `DiscountDelegate` objects, add statements that assign the `firstDel` object to `thirdDel`, and then add `secondDel` to it through composition.

```
thirdDel = firstDel;  
thirdDel += secondDel;
```

4. Change the prompt for the code, as follows, to reflect three options. The standard and preferred discounts remain the same, but the extreme discount (supposedly for special customers) provides both types of discounts, first subtracting 5 percent for any sale equal to or greater than \$1,000, and then providing a discount of 10 percent more.

```
Write("Enter S for standard discount, " +  
      "P for preferred discount," +  
      "\nor X for extreme discount ");
```

5. Change the `if` statement so that if the user does not enter `S` or `P`, then the extreme discount applies.

```
if(code == 'S')  
    firstDel(ref saleAmount);  
else  
    if(code == 'P')  
        secondDel(ref saleAmount);  
    else  
        thirdDel(ref saleAmount);
```

(continues)

(continued)

- Save the program, and then compile and execute it. For reference, Figure 13-11 shows the complete program. Figure 13-12 shows the output when the program is executed several times. When the user enters a sale amount of \$1,000 and an S, a 5 percent discount is applied. When the user enters a P for the same amount, a 10 percent discount is applied. When the user enters X with the same amount, a 5 percent discount is applied, followed by a 10 percent discount, which produces a net result of a 14.5 percent discount.

```

using System;
using static System.Console;
delegate void DiscountDelegate(ref double saleAmount);
class DiscountDelegateDemo2
{
    public static void StandardDiscount(ref double saleAmount)
    {
        const double DISCOUNT_RATE = 0.05;
        const double CUTOFF = 1000.00;
        double discount;
        if(saleAmount >= CUTOFF)
            discount = saleAmount * DISCOUNT_RATE;
        else
            discount = 0;
        saleAmount -= discount;
    }
    public static void PreferredDiscount(ref double saleAmount)
    {
        const double SPECIAL_DISCOUNT = 0.10;
        double discount = saleAmount * SPECIAL_DISCOUNT;
        saleAmount -= discount;
    }
    static void Main()
    {
        double saleAmount;
        char code;
        DiscountDelegate firstDel, secondDel, thirdDel;
        firstDel = new DiscountDelegate(StandardDiscount);
        secondDel = new DiscountDelegate(PreferredDiscount);
        thirdDel = firstDel;
        thirdDel += secondDel;
        Write("Enter amount of sale ");
    }
}

```

Figure 13-11 The DiscountDelegateDemo2 program (*continues*)

(continues)

(continued)

(continued)

```

saleAmount = Convert.ToDouble(ReadLine());
Write("Enter S for standard discount, " +
      "P for preferred discount, " +
      "\nor X for eXtreme discount ");
code = Convert.ToChar(ReadLine());
if(code == 'S')
    firstDel(ref saleAmount);
else
    if(code == 'P')
        secondDel(ref saleAmount);
    else
        thirdDel(ref saleAmount);
WriteLine("New sale amount is {0}",
          saleAmount.ToString("C2"));
}
}

```

Figure 13-11 The DiscountDelegateDemo2 program

```

C:\C#\Chapter.13>DiscountDelegateDemo2
Enter amount of sale 1000
Enter S for standard discount, P for preferred discount,
or X for eXtreme discount S
New sale amount is $950.00

C:\C#\Chapter.13>DiscountDelegateDemo2
Enter amount of sale 1000
Enter S for standard discount, P for preferred discount,
or X for eXtreme discount P
New sale amount is $900.00

C:\C#\Chapter.13>DiscountDelegateDemo2
Enter amount of sale 1000
Enter S for standard discount, P for preferred discount,
or X for eXtreme discount X
New sale amount is $855.00

C:\C#\Chapter.13>

```

Figure 13-12 Three executions of the DiscountDelegateDemo2 program

For static methods like `StandardDiscount` and `PreferredDiscount`, a delegate object encapsulates the method to be called. When creating a class that contains instance methods, you create delegate objects that encapsulate both an instance of the class and a method of the instance. You will create this type of delegate in the next section.

Declaring Your Own Events and Handlers and Using the Built-in EventHandler

622

To declare your own event, you use a delegate. An event provides a way for the clients of a class to dictate methods that should execute when an event occurs. The clients identify methods to execute by associating the delegate with the method that should execute when the event occurs. Just like the event handlers that are automatically created in the IDE, each of your own event handler delegates requires two arguments: the object where the event was initiated (the sender) and an EventArgs argument. You can create an EventArgs object that contains event information, or you can use the EventArgs class static field named Empty, which represents an event that contains no event data. In other words, using the EventArgs.Empty field simply tells the client that an event has occurred, without specifying details. For example, you can declare a delegate event handler named ChangedEventHandler, as follows:

```
public delegate void ChangedEventHandler  
    (object sender, EventArgs e);
```

The identifier ChangedEventHandler can be any legal identifier you choose. This delegate defines the set of arguments that will be passed to the method that handles the event. The delegate ChangedEventHandler can be used in a client program that handles events.

For example, Figure 13-13 contains the ChangedEventHandler delegate and a simple Student class that is similar to many classes you already have created. The Student class contains just two data fields and will generate an event when the data in either field changes.

```
public delegate void ChangedEventHandler(object sender, EventArgs e);  
class Student  
{  
    private int idNum;  
    private double gpa;  
    public event ChangedEventHandler Changed;  
    public int IdNum  
    {  
        get  
        {  
            return idNum;  
        }  
        set  
        {  
            idNum = value;  
            OnChanged(EventArgs.Empty);  
        }  
    }  
}
```

Figure 13-13 The Student class (continues)

(continued)

```

public double Gpa
{
    get
    {
        return gpa;
    }
    set
    {
        gpa = value;
        OnChanged(EventArgs.Empty);
    }
}
private void OnChanged(EventArgs e)
{
    Changed(this, e);
}
}

```

Figure 13-13 The Student class

The class in Figure 13-13 contains fields that hold an ID number and grade point average for a `Student`. The first shaded statement in the figure defines a third attribute of the `Student` class—an event named `Changed`. The declaration for an event looks like a field, but instead of being an `int` or a `double`, it is a `ChangedEventHandler`.



Events usually are declared as `public`, but you can use any accessibility modifier.

The `Student` class event (`Changed`) looks like an ordinary field. However, you cannot assign values to the event as easily as you can to ordinary data fields. You can take only two actions on an event: You can compose a new delegate onto it using the `+=` operator, and you can remove a delegate from it using the `-=` operator. For example, to add `StudentChanged` to the `Changed` event of a `Student` object named `stu`, you would write the following:

```
stu.Changed += new ChangedEventHandler(StudentChanged);
```

In the `Student` class, each `set` accessor assigns a value to the appropriate class instance field. However, when either `idNum` or `gpa` changes, the method in the `Student` class named `OnChanged()` is also called, using `EventArgs.Empty` as the argument. The value of `EventArgs.Empty` is a read-only instance of `EventArgs`. You can pass it to any method that accepts an `EventArgs` parameter.

The `OnChanged()` method calls `Changed()` using two arguments: a reference to the `Student` object that was changed and the empty `EventArgs` object. Calling `Changed()` is also known as **invoking the event**.



If no client has wired a delegate to the event, the `Changed` field will be `null`, rather than referring to the delegate that should be called when the event is invoked. Therefore, programmers often check for `null` before invoking the event, as in the following example:

```
if(Changed != null)
    Changed(this, e);
```

For simplicity, the example in Figure 13-13 does not bother checking for `null`.

Figure 13-14 shows an `EventListener` class that listens for `Student` events. This class contains a `Student` object that is assigned a value using the parameter to the `EventListener` class constructor. The `StudentChanged()` method is added to the `Student`'s event delegate using the `+=` operator. The `StudentChanged()` method is the event-handler method that executes in response to a `Changed` event; it displays a message and `Student` data.

```
class EventListener
{
    private Student stu;
    public EventListener(Student student)
    {
        stu = student;
        stu.Changed += new ChangedEventHandler(StudentChanged);
    }
    private void StudentChanged(object sender, EventArgs e)
    {
        WriteLine("The student has changed.");
        WriteLine("ID# {0} GPA {1}",
            stu.IdNum, stu.Gpa);
    }
}
```

Figure 13-14 The `EventListener` class

Figure 13-15 shows a program that demonstrates using the `Student` and `EventListener` classes. The program contains a single `Main()` method, which declares a `Student` and an `EventListener` that listens for events from the `Student` class. Then three assignments are made. Because this program is registered to listen for events from the `Student`, each change in a data field triggers an event. That is, each field assignment not only changes the value of the data field, it also executes the `StudentChanged()` method that displays two lines of explanation. In Figure 13-16, the program output shows that an event occurs three times—once when `IdNum` becomes 2345 (and `Gpa` is still 0), again when `IdNum` becomes 4567 (and `Gpa` still has not changed), and a third time when `Gpa` becomes 3.2.

```
using System;
using static System.Console;
class DemoStudentEvent
{
    static void Main()
    {
        Student oneStu = new Student();
        EventListener listener = new EventListener(oneStu);
        oneStu.IdNum = 2345;
        oneStu.IdNum = 4567;
        oneStu.Gpa = 3.2;
    }
}
```

Figure 13-15 The DemoStudentEvent program

```
C:\C#\Chapter.13>DemoStudentEvent
The student has changed.
  ID# 2345  GPA 0
The student has changed.
  ID# 4567  GPA 0
The student has changed.
  ID# 4567  GPA 3.2
C:\C#\Chapter.13>
```

Figure 13-16 Output of the DemoStudentEvent program

Using the Built-in EventHandler

The C# language allows you to create events using any delegate type. However, the .NET Framework provides guidelines you should follow if you are developing a class that others will use. These guidelines indicate that the delegate type for an event should take exactly two parameters: a parameter indicating the source of the event, and an `EventArgs` parameter that encapsulates any additional information about the event. For events that do not use additional information, the .NET Framework has already defined an appropriate delegate type named `EventHandler`.

Figure 13-17 shows all the code necessary to demonstrate an `EventHandler`. Note the following changes from the classes used in the `DemoStudentEvent` program:

- No delegate is declared explicitly in the `Student` class.
- In the first statement with shading in the `Student` class, the event is associated with the built-in delegate `EventHandler`.
- In the second statement with shading, which appears in the `EventListener` class, the delegate composition uses `EventHandler`.

```
using System;
using static System.Console;
class Student
{
    private int idNum;
    private double gpa;
    public event EventHandler Changed;
    public int IdNum
    {
        get
        {
            return idNum;
        }
        set
        {
            idNum = value;
            OnChanged(EventArgs.Empty);
        }
    }
    public double Gpa
    {
        get
        {
            return gpa;
        }
        set
        {
            gpa = value;
            OnChanged(EventArgs.Empty);
        }
    }
    private void OnChanged(EventArgs e)
    {
        Changed(this, e);
    }
}
class EventListener
{
    private Student stu;
    public EventListener(Student student)
    {
        stu = student;
        stu.Changed += new EventHandler(StudentChanged);
    }
}
```

Figure 13-17 The Student, EventListener, and DemoStudentEvent2 classes (*continues*)

(continued)

```

private void StudentChanged(object sender, EventArgs e)
{
    WriteLine("The student has changed.");
    WriteLine("  ID# {0} GPA {1}"
        stu.IdNum, stu.Gpa);
}
}
class DemoStudentEvent2
{
    static void Main()
    {
        Student oneStu = new Student();
        EventListener listener = new EventListener(oneStu);
        oneStu.IdNum = 2345;
        oneStu.IdNum = 4567;
        oneStu.Gpa = 3.2;
    }
}

```

Figure 13-17 The Student, EventListener, and DemoStudentEvent2 classes

When you compile and execute the program in Figure 13-17, the output is identical to that shown in Figure 13-16.

TWO TRUTHS & A LIE

Declaring Your Own Events and Handlers and Using the Built-in EventHandler

1. When an event occurs, any delegate that a client has given or passed to the event is invoked.
2. Built-in event handler delegates have two arguments, but those you create yourself have only one.
3. You can take only two actions on an event field: composing a new delegate onto the field using the += operator, and removing a delegate from the field using the -= operator.

The false statement is #2. Every event handler delegate requires two arguments—the object where the event was initiated (the sender) and an EventArgs argument.



You Do It

628

Creating a Delegate That Encapsulates Instance Methods

In the next set of steps, you create a simple `BankAccount` class that contains just two data fields: an account number and a balance. It also contains methods to make withdrawals and deposits. An event is generated after any withdrawal or deposit.

1. Open a project named **DemoBankEvent**. Type the `using` statements you need, and then begin a class named `BankAccount`. The class contains an account number, a balance, and an event that executes when an account's balance is adjusted.

```
using System
using static System.Console;
class BankAccount
{
    private int acctNum;
    private double balance;
    public event EventHandler BalanceAdjusted;
```

2. Add a constructor that accepts an account number parameter and initializes the balance to 0.

```
public BankAccount(int acct)
{
    acctNum = acct;
    balance = 0;
}
```

3. Add read-only properties for both the account number and the account balance.

```
public int AcctNum
{
    get
    {
        return acctNum;
    }
}
public double Balance
{
    get
    {
        return balance;
    }
}
```

(continues)

(continued)

4. Add two methods. One makes account deposits by adding the parameter to the account balance, and the other makes withdrawals by subtracting the parameter value from the bank balance. Each uses the `OnBalanceAdjusted` event handler that reacts to all deposit and withdrawal events by displaying the new balance.

```
public void MakeDeposit(double amt)
{
    balance += amt;
    OnBalanceAdjusted(EventArgs.Empty);
}
public void MakeWithdrawal(double amt)
{
    balance -= amt;
    OnBalanceAdjusted(EventArgs.Empty);
}
```

5. Add the `OnBalanceAdjusted()` method that accepts an `EventArgs` parameter and calls `BalanceAdjusted()`, passing it references to the newly adjusted `BankAccount` object and the `EventArgs` object. (Earlier in the chapter, you learned that calling a method such as `OnBalanceAdjusted()` is also known as *invoking the event*.) Include a closing curly brace for the class.

```
public void OnBalanceAdjusted(EventArgs e)
{
    BalanceAdjusted(this, e);
}
}
```

6. Save the file.

Creating an Event Listener

1. When you write an application that declares a `BankAccount`, you might want the client program to listen for `BankAccount` events. To do so, you create an `EventListener` class.
2. After the closing curly brace of the `BankAccount` class, type the following `EventListener` class that contains a `BankAccount` object. When the `EventListener` constructor executes, the `BankAccount` field is initialized with the constructor parameter. Using the `+=` operator, add the `BankAccountBalanceAdjusted()` method to the event delegate. Next, write the `BankAccountBalanceAdjusted()` method to display a message and information about the `BankAccount`.

(continues)

(continued)

```

class EventListener
{
    private BankAccount acct;
    public EventListener(BankAccount account)
    {
        acct = account;
        acct.BalanceAdjusted += new EventHandler
            (BankAccountBalanceAdjusted);
    }
    private void BankAccountBalanceAdjusted(object sender,
        EventArgs e)
    {
        WriteLine("The account balance has been adjusted.");
        WriteLine(" Account# {0} balance {1}",
            acct.AcctNum, acct.Balance.ToString("C2"));
    }
}

```

3. Create a class to test the `BankAccount` and `EventListener` classes. Below the closing curly brace for the `EventListener` class, start a `DemoBankAccountEvent` class that contains a `Main()` method. Declare an integer to hold the number of transactions that will occur in the demonstration program. Also declare two variables: one can hold a code that indicates whether a transaction is a deposit or withdrawal, and one is the amount of the transaction.

```

class DemoBankAccountEvent
{
    static void Main()
    {
        const int TRANSACTIONS = 5;
        char code;
        double amt;
    }
}

```

4. Declare a `BankAccount` object that is assigned an arbitrary account number, and declare an `EventListener` object so this program is registered to listen for events from the `BankAccount`. Each change in the `BankAccount` balance will not only change the balance data field, it will execute the `BankAccountBalanceAdjusted()` method that displays two lines of explanation.

```

BankAccount acct = new BankAccount(334455);
EventListener listener = new EventListener (acct);

```

(continues)

(continued)

5. Add a loop that executes five times (the value of `TRANSACTIONS`). On each iteration, prompt the user to indicate whether the current transaction is a deposit or withdrawal and to enter the transaction amount. Call the `MakeDeposit()` or `MakeWithdrawal()` method accordingly.

```
for(int x = 0; x < TRANSACTIONS; ++x)
{
    Write("Enter D for deposit or W for withdrawal ");
    code = Convert.ToChar(ReadLine());
    Write("Enter dollar amount ");
    amt = Convert.ToDouble(ReadLine());
    if(code == 'D')
        acct.MakeDeposit(amt);
    else
        acct.MakeWithdrawal(amt);
}
```

6. At the end of the `for` loop, add a closing curly brace for the `Main()` method and another one for the class.
7. Save the file, and then compile and execute it. Figure 13-18 shows a typical execution in which five transactions modify the account. The output shows that an event occurs five times—twice for deposits and three times for withdrawals.

```
C:\C#\Chapter.13>DemoBankEvent
Enter D for deposit or W for withdrawal D
Enter dollar amount 2000
The account balance has been adjusted.
Account# 334455 balance $2,000.00
Enter D for deposit or W for withdrawal W
Enter dollar amount 250
The account balance has been adjusted.
Account# 334455 balance $1,750.00
Enter D for deposit or W for withdrawal W
Enter dollar amount 600
The account balance has been adjusted.
Account# 334455 balance $1,150.00
Enter D for deposit or W for withdrawal D
Enter dollar amount 500
The account balance has been adjusted.
Account# 334455 balance $1,650.00
Enter D for deposit or W for withdrawal W
Enter dollar amount 825
The account balance has been adjusted.
Account# 334455 balance $825.00

C:\C#\Chapter.13>
```

Figure 13-18 Typical execution of the `DemoBankEvent` program

Handling Control Component Events

Handling events requires understanding several difficult concepts. Fortunately, you most frequently will want to handle events in GUI environments when the user will manipulate Controls, and the good news is that these events have already been defined for you. When you want to handle events generated by GUI Controls, you use the same techniques as when you handle any other events. The major difference is that when you create your own classes, like `Student`, you must define both the data fields and events you want to manage. However, existing Control components, like `Buttons` and `ListBoxes`, already contain fields and `public` properties, like `Text`, as well as events with names, like `Click`. Table 13-1 lists just some of the more commonly used Control events. You can consult the Visual Studio Help feature to discover additional Control events as well as more specific events assigned to individual Control child classes.

Event	Description
<code>BackColorChanged</code>	Occurs when the value of the <code>BackColor</code> property has changed
<code>Click</code>	Occurs when a control is clicked
<code>ControlAdded</code>	Occurs when a new control is added
<code>ControlRemoved</code>	Occurs when a control is removed
<code>CursorChanged</code>	Occurs when the <code>Cursor</code> property value has changed
<code>DragDrop</code>	Occurs when a drag-and-drop operation is completed
<code>DragEnter</code>	Occurs when an object is dragged into a control's bounds
<code>DragLeave</code>	Occurs when an object has been dragged into and out of a control's bounds
<code>DragOver</code>	Occurs when an object has been dragged over a control's bounds
<code>EnabledChanged</code>	Occurs when the <code>Enabled</code> property value has changed
<code>Enter</code>	Occurs when a control is entered
<code>FontChanged</code>	Occurs when the <code>Font</code> property value has changed
<code>ForeColorChanged</code>	Occurs when the <code>ForeColor</code> property value has changed
<code>GotFocus</code>	Occurs when a control receives focus
<code>HelpRequested</code>	Occurs when a user requests help for a control
<code>KeyDown</code>	Occurs when a key is pressed while a control has focus; event is followed immediately by <code>KeyPress</code>
<code>KeyPress</code>	Occurs when a key is pressed while a control has focus; event occurs just after <code>KeyDown</code>

Table 13-1 Some Control class `public` instance events (*continues*)

(continued)

Event	Description
KeyUp	Occurs when a key is released while a control has focus
Leave	Occurs when a control is left
LocationChanged	Occurs when the <code>Location</code> property value has changed
LostFocus	Occurs when a control loses focus
MouseDown	Occurs when the mouse pointer hovers over a control and a mouse button is pressed
MouseEnter	Occurs when the mouse pointer enters a control
MouseHover	Occurs when the mouse pointer hovers over a control
MouseLeave	Occurs when the mouse pointer leaves a control
MouseMove	Occurs when the mouse pointer moves over a control
MouseUp	Occurs when the mouse pointer hovers over a control and a mouse button is released
MouseWheel	Occurs when the mouse wheel moves while a control has focus
Move	Occurs when a control is moved
Resize	Occurs when a control is resized
TextChanged	Occurs when the <code>Text</code> property value has changed
VisibleChanged	Occurs when the <code>Visible</code> property value has changed

Table 13-1 Some Control class public instance events

You have already used the IDE to create some event-handling methods. These methods have been the default events generated when you double-click a Control in the IDE. For example, you have created a `Click()` method for a `Button` and a `LinkClicked()` method for a `LinkLabel`. A `Form` can contain any number of Controls that might have events associated with them. Additionally, a single Control might be able to raise any number of events. For example, besides creating a Button's default `Click` event, you might want to define various actions when the user's mouse rolls over the button. Table 13-1 lists only a few of the many events available with Controls; any Control could conceivably raise many of the available events.

Suppose you want to create a project that takes a different set of actions when the mouse is over a `Button` than when the mouse is clicked. Figure 13-19 shows a project that has been started in the IDE. The following actions have been taken:

- The `Form` was resized to 225, 150.
- A `Button` was dragged onto the `Form`, and its `Text` was set to *Click me*.
- A `Label` was added to the `Form`, its `Text` was set to *Hello*, and its `Font` was increased.

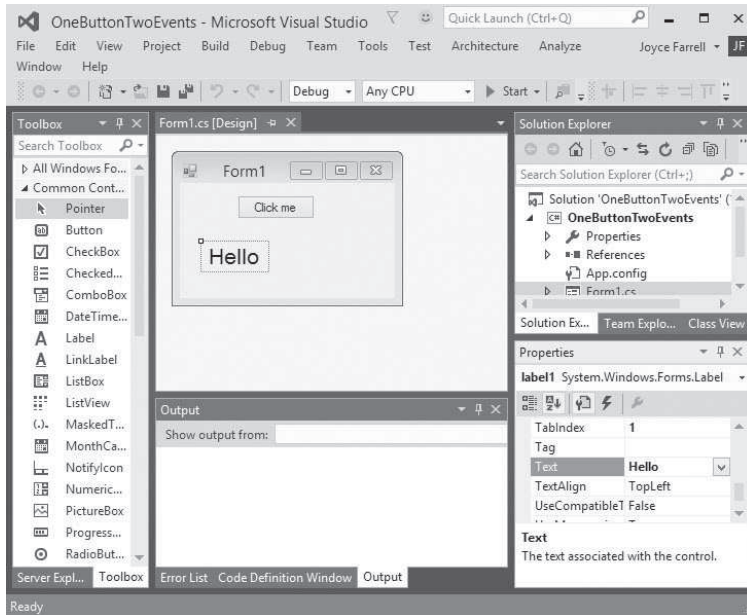


Figure 13-19 Start of the OneButtonTwoEvents project in the IDE

When you double-click the `Button` on the `Form` in the IDE, you generate the shell of a `Click()` method into which you can type a command to change the `Label`'s text and color, as follows:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = "Button was clicked";
    label1.BackColor = Color.CornflowerBlue;
}
```



`Color.CornflowerBlue` is one of C#'s predefined `Color` properties. A complete list appears in Table 12-5 in Chapter 12.

With the `Button` selected on the design `Form`, you can click the Events icon in the Properties window at the right side of the screen. The Events icon looks like a lightning bolt. Figure 13-20 shows that the Properties window displays events instead of properties and that the `Click` event has an associated method.

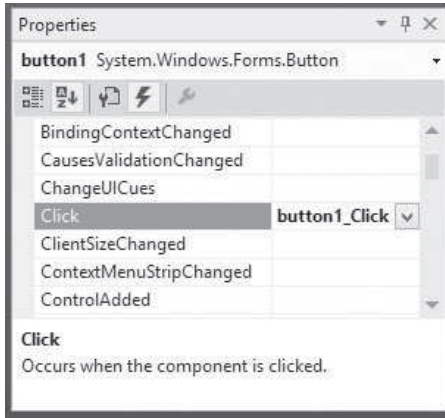


Figure 13-20 The Properties window displaying events

If you scroll through the Events listed in the Properties window, you can see a wide variety of Event choices. If you scroll down to `MouseEnter` and double-click, you can view the code for an event handler in the `Form1.cs` file as follows:

```
private void button1_MouseEnter(object sender, EventArgs e)
{
}
```



When you are viewing events in the Properties window, you can return to the list of properties by clicking the Properties icon. This icon is to the immediate left of the Events icon.

You can type any statements you want within this method. For example, you might write the following method:

```
private void button1_MouseEnter(object sender, EventArgs e)
{
    label1.Text = "Go ahead";
    button1.BackColor = Color.Red;
}
```

When you run the program with the two new methods, two different events can occur:

- When you enter the button with the mouse (that is, pass the mouse over it), the `Label`'s `Text` changes to *Go ahead*, and the button turns red, as shown on the left in Figure 13-21.
- After the button is clicked, the `Label`'s `Text` changes again to *Button was clicked*, and the `Label` becomes blue, as shown on the right in Figure 13-21.

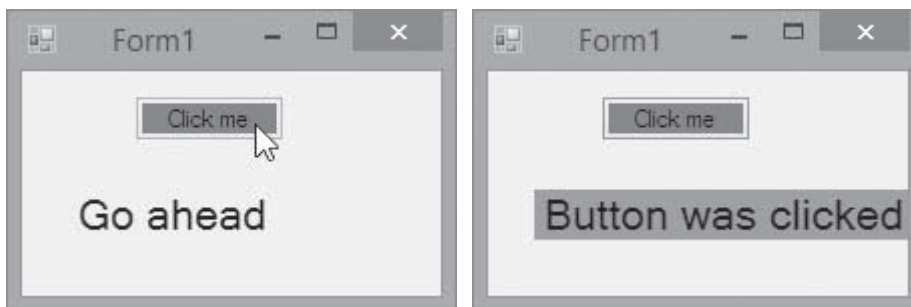


Figure 13-21 The `OneButtonTwoEvents` program when the mouse enters the button and after the button is clicked

If you examine the code generated by the Windows Form Designer, you will find the following two statements:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
this.button1.MouseEnter += new
    System.EventHandler(this.button1_MouseEnter);
```

636

These `EventHandler` statements are similar to those in the `Student` class in Figure 13-17. The `Click` and `MouseEnter` delegates have been set to handle events appropriately for this application. You could have used the IDE to create these events just by selecting them from the Properties list and writing the action statements you want. The IDE saves you time by automatically entering the needed statement correctly. However, by knowing how to manually create a GUI program that contains events, you gain a greater understanding of how event handling works. This knowledge helps you troubleshoot problems and helps you create your own new events and handlers when necessary.



Watch the video *Handling Control Component Events*.

TWO TRUTHS & A LIE

Handling Control Component Events

1. The default methods generated when you double-click a `Control` in the IDE are known as procedures.
2. A `Form` can contain any number of `Controls` that might have events associated with them, and a single `Control` might be able to raise any number of events.
3. You can type any statements you want within an automatically generated event method.

The false statement is #1. The default methods generated when you double-click a `Control` in the IDE are known as event handlers.

Handling Mouse and Keyboard Events

Users can interact with GUI applications in multiple ways. Certainly the most common tactics are to use a mouse or to type on a keyboard. Mouse and keyboard events are similar in that every mouse or key event-handling method must have two parameters: an object representing the sender and an object that holds information about the event.

Handling Mouse Events

Mouse events include all the actions a user takes with a mouse, including clicking, pointing, and dragging. Mouse events can be handled for any `Control` through an object of the class `MouseEventArgs`, which descends from `EventArgs`. The delegate used to create mouse event handlers is `EventHandler`. Depending on the event, the type of the second parameter in the event-handling method is `EventArgs` or `MouseEventArgs`. Table 13-2 describes several common mouse events, and Table 13-3 lists some properties of the `MouseEventArgs` class.

Mouse event	Description	Event argument type
<code>MouseClicked</code>	Occurs when the user clicks the mouse within the <code>Control</code> 's boundaries	<code>MouseEventArgs</code>
<code>MouseDoubleClick</code>	Occurs when the user double-clicks the mouse within the <code>Control</code> 's boundaries	<code>MouseEventArgs</code>
<code>MouseEnter</code>	Occurs when the mouse cursor enters the <code>Control</code> 's boundaries	<code>EventArgs</code>
<code>MouseLeave</code>	Occurs when the mouse cursor leaves the <code>Control</code> 's boundaries	<code>EventArgs</code>
<code>MouseDown</code>	Occurs when a mouse button is pressed while the mouse is within the <code>Control</code> 's boundaries	<code>MouseEventArgs</code>
<code>MouseHover</code>	Occurs when the mouse cursor is within the <code>Control</code> 's boundaries	<code>MouseEventArgs</code>
<code>MouseMove</code>	Occurs when the mouse is moved while within the <code>Control</code> 's boundaries	<code>MouseEventArgs</code>
<code>MouseUp</code>	Occurs when a mouse button is released while the mouse is within the <code>Control</code> 's boundaries	<code>MouseEventArgs</code>

Table 13-2 Common mouse events



A `MouseDown` event can occur without a corresponding `MouseUp` if the user presses the mouse but switches focus to another control or application before releasing the mouse button.

MouseEventArgs property	Description
Button	Specifies which mouse button triggered the event; the value can be <code>Left</code> , <code>Right</code> , <code>Middle</code> , or <code>none</code>
Clicks	Specifies the number of times the mouse was clicked
X	The x-coordinate where the event occurred on the control that generated the event
Y	The y-coordinate where the event occurred on the control that generated the event

Table 13-3 Properties of the `MouseEventArgs` class



`MouseClicked` and `Click` are separate events. The `Click` event takes an `EventArgs` parameter, but `MouseClicked` takes a `MouseEventArgs` parameter. For example, if you define a `Click` event, you do not have the `MouseEventArgs` class properties.

Each part of Figure 13-22 contains a `Form` with a single `Label` named `clickLocationLabel` that changes as the user continues to click the mouse on it. The figure shows how the `Label` changes in response to a series of user clicks. Initially, the `Label` is empty (that is, the default `Text` property has been deleted and not replaced), but the following code was added to the `Form.cs` file:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    clickLocationLabel.Text += "\nClicked at " + e.X +
        ", " + e.Y;
}
```

Every time the mouse is clicked on the `Form`, the `Label` is appended with a new line that contains “Clicked at” and the x- and y-coordinate position where the click occurred on the `Form`.

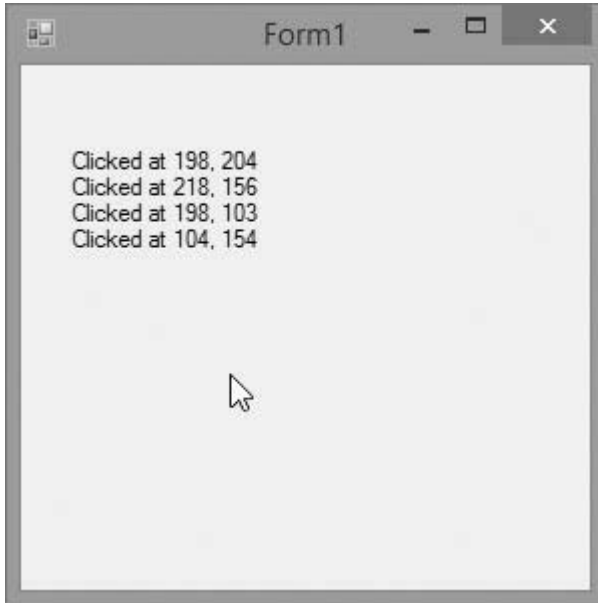


Figure 13-22 A Form that responds to clicks

When the programmer selects the `MouseClicked()` event for `Form1` from the Event list in the IDE, the following code is generated in the `Designer.cs` file. This code instantiates the `MouseClicked` delegate.

```
this.MouseClick += new System.Windows.Forms.MouseEventHandler
    (this.Form1_MouseClicked);
```

Handling Keyboard Events

Keyboard events, also known as **key events**, occur when a user presses and releases keyboard keys. Table 13-4 lists some common keyboard events. Similar to the way mouse events work, every keyboard event-handling method must have two parameters. Depending on the event, the delegate used to create the keyboard event handler is either `KeyEventHandler` or `KeyPressEventHandler` and the type of the second parameter is `EventArgs` or `KeyPressEventArgs`.

Keyboard event	Description	Event argument type
<code>KeyDown</code>	Occurs when a key is first pressed	<code>EventArgs</code>
<code>KeyUp</code>	Occurs when a key is released	<code>EventArgs</code>
<code>KeyPress</code>	Occurs when a key is pressed	<code>KeyPressEventArgs</code>

Table 13-4 Keyboard events

Table 13-5 describes `KeyEventArgs` properties, and Table 13-6 describes `KeyPressEventArgs` properties. An important difference is that `KeyEventArgs` objects include data about helper keys or modifier keys that are pressed with another key. For example, if you need to distinguish between a user pressing *A* and pressing *Alt+A* in your application, then you must use a keyboard event that uses an argument of type `KeyEventArgs`.

Property	Description
<code>Alt</code>	Indicates whether the Alt key was pressed
<code>Control</code>	Indicates whether the Control (Ctrl) key was pressed
<code>Shift</code>	Indicates whether the Shift key was pressed
<code>KeyCode</code>	Returns the code for the key
<code>KeyData</code>	Returns the key code along with any modifier key
<code>KeyValue</code>	Returns a numeric representation of the key (this number is known as the Windows virtual key code)

Table 13-5 Some properties of `KeyEventArgs` class

Property	Description
<code>KeyChar</code>	Returns the ASCII character for the key pressed

Table 13-6 A property of `KeyPressEventArgs` class

For example, suppose that you create a `Form` with a `Label`, like the `Form` in Figure 13-24. From the Properties window for the `Form`, you can double-click the `KeyUp` event to generate the shell for a method named `Form1_KeyUp()`. Suppose you then insert the statements into the method in the `Form1.cs` file, as shown in Figure 13-23. When the user runs the program and presses and releases a keyboard key, the `Label` is filled with information about the key. Figure 13-24 shows three executions of this modified program. During the first execution, the user typed *a*. You can see on the form that the `KeyCode` is *A* (uppercase), but you also can see that the user did not press the Shift key.

```
private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    label1.Text += "Key Code " + e.KeyCode;
    label1.Text += "\nAlt " + e.Alt;
    label1.Text += "\nShift " + e.Shift;
    label1.Text += "\nControl " + e.Control;
    label1.Text += "\nKey Data " + e.KeyData;
    label1.Text += "\nKey Value " + e.KeyValue + "\n\n";
}
```

Figure 13-23 The `KeyUp()` method

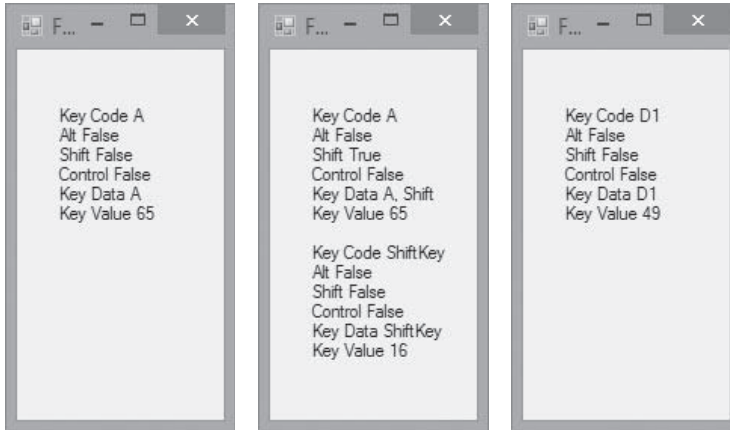


Figure 13-24 Three executions of the KeyDemo program

In the second execution in Figure 13-24, the user held down Shift, pressed *a*, and then released the Shift key. This causes two separate `KeyUp` events. The first has `KeyCode A` with `Shift true`, and the second has `KeyCode ShiftKey`. Notice that the Key values generated after typing *a* and *A* are the same, but the value after typing Shift is different.

In the third execution in Figure 13-24, the user pressed the number *1*, whose code is `D1`.

When you view the `Form1.Designer.cs` file for the KeyDemo program, you see the following automatically created statement, which defines the composed delegate:

```
this.KeyUp += new System.Windows.Forms.KeyEventHandler
    (this.Form1_KeyUp);
```

TWO TRUTHS & A LIE

Handling Mouse and Keyboard Events

1. The delegate used to create mouse event handlers is `MouseEventHandler`.
2. Keyboard events, also known as key events, occur when a user presses and releases keyboard keys.
3. Unlike mouse events, every keyboard event-handling method must be parameterless.

The false statement is #3. Like mouse events, every keyboard event-handling method must have two parameters: an object representing the sender, and an object that holds information about the event.

Managing Multiple Controls

When Forms contain multiple Controls, you often want one to have focus. You also might want several actions to have a single, shared consequence. For example, you might want the same action to occur whether the user clicks a button or presses the Enter key, or you might want multiple buttons to generate the same event when they are clicked.

642

Defining Focus

When users encounter multiple GUI Controls on a Form, usually one Control has **focus**. For example, when a Button has focus, a user expects that clicking the button or pressing the Enter key will raise an event.

TabStop is a Boolean property of a Control that identifies whether the Control will serve as a stopping place—that is, a place that will receive focus—in a sequence of Tab key presses. The default value for **TabStop** for a Button is **true**, but the default value for a Label is **false** because Labels are not expected to be part of a tabbing sequence. **TabIndex** is a numeric property that indicates the order in which the Control will receive focus when the user presses the Tab key. Programmers typically use small numbers for **TabIndex** values, beginning with 0. When a Control's **TabStop** value is **true** and the Control has the lowest **TabIndex** of a Form's Controls, it receives focus when the Form is initialized.



Setting the **TabIndex** values of two or more Controls to 0 does not cause an error. Only one Control will receive focus, however.

Figure 13-25 shows a Form that contains three Buttons and a Label. Each Button has been associated with a `Click()` event such as the following:

```
private void button1_Click(object sender, EventArgs e)
{
    buttonInfoLabel.Text = "Button 1 selected";
}
```

Each Button has been assigned a **TabIndex** value in ascending order. When the application starts, the first Button (labeled 1) has focus. Whether the user clicks that button or presses Enter, the message appears as shown on the left in Figure 13-25. In the next part of the figure, the user has pressed Tab and Enter to select `button2`, so it has focus, and the Label's **Text** property has changed. Alternatively, the user could have clicked `button2` to achieve the same result. The user could then select either of the other Buttons by clicking them as usual or by pressing Tab until the desired button has focus and then pressing Enter.

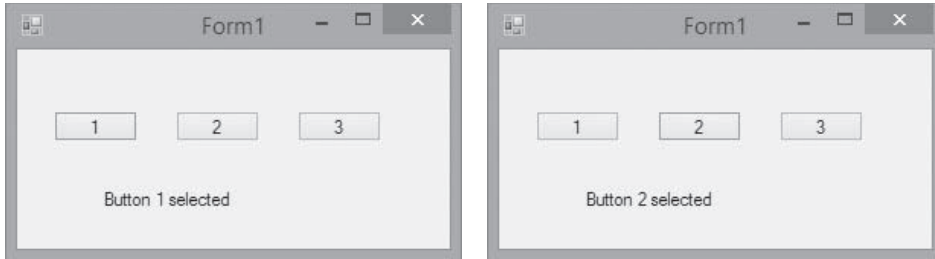


Figure 13-25 Execution of the FocusDemo application

Handling Multiple Events with a Single Handler

When a Form contains multiple Controls, you can create a separate event handler for each Control. However, you can also associate the same event handler with multiple Controls. For example, suppose you create a Form that contains three Buttons and a Label. The buttons have been labeled *A*, *B*, and *3*. Further suppose you want to display one message when the user clicks a letter button and a different message when the user clicks a number button. In the IDE, you can double-click the first Button and create an associated method such as the following:

```
private void button1_Click(object sender, EventArgs e)
{
    buttonInfoLabel.Text = "You clicked a letter button";
}
```

If you click the second button so its properties are displayed in the IDE's Properties list, you can click the Events icon to see a list of events associated with the second button. If no Click event has been chosen yet, a list box is available. This list contains all the existing events that have the correct signature to be the event handler for the event. Because the `button1_Click()` handler can also handle a `button2_Click()` event, you can select it as the event for `button2`. When you run the program, clicking either letter button produces the output shown in Figure 13-26.

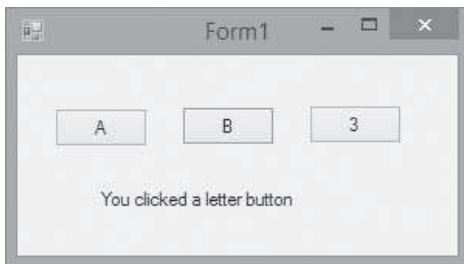


Figure 13-26 Output of the SingleHandler program after a letter button is clicked



Until you associate an event with the *3* button, nothing happens when the user clicks it. To complete the application that is running in Figure 13-26, you would want to associate an event with the *3* button to modify the Label's Text to *You clicked a number button*.

When two or more Controls generate the same event, many programmers prefer to generalize the event method name. For example, if `button1` and `button2` call the same method when clicked, it makes sense to name the event method `letterButton_Click()` instead of `button1_Click()`.

Perhaps you have shopped online at a site that offers multiple ways to “buy now.” For example, you might click a grocery cart icon, choose *Place order now* from a menu, or click a button. Often, *Place order now* buttons are displayed at several locations on the page. If you want to encourage a user’s behavior, you should provide multiple ways to accommodate it.



Watch the video *Managing Multiple Controls*.

TWO TRUTHS & A LIE

Managing Multiple Controls

1. The `Control.TabStop` property can be set to `true` or `false`; it identifies whether the `Control` will serve as a stopping place in a sequence of `Tab` key presses.
2. On a `Form` with multiple `Controls`, one `Control` must have a `TabIndex` value of 0.
3. When a `Form` contains multiple `Controls`, you can associate the same event with all of them.

The false statement is #2. `TabIndex` is a numeric property that indicates the order in which the control will receive focus when the user presses the `Tab` key. Programmers typically use small numbers for `TabIndex` values, beginning with 0. However, you might choose not to use any `TabIndex` values, or if you do, you might choose not to start with 0.

Continuing to Learn about Controls and Events

If you examine the Visual Studio IDE, you will discover many additional `Controls` that contain hundreds of properties and events. No single book or programming course can demonstrate all of them for you. However, if you understand good programming principles and the syntax and structure of `C#` programs, learning about each new `C#` feature becomes progressively easier. When you encounter a new control in the IDE, you probably can use it without understanding all the code generated in the background, but when you do understand the background, your knowledge of `C#` is more complete.

Continue to explore the Help facility in the Visual Studio IDE. Particularly, read the brief tutorials there. Also, you should search the Internet for `C#` discussion groups. `C#` is a new, dynamic language, and programmers pose many questions to each other online. Reading these discussions can provide you with valuable information and suggest new approaches to resolving problems.



You Do It

Using `TabIndex`

In the next steps, you create a Form in the Visual Studio IDE and add four Buttons so you can demonstrate how to manipulate their `TabIndex` properties.

1. Open the Visual Studio IDE and start a new project. Define it to be a **Windows Forms Application** named **ManyButtons**.
2. Change the Size of the Form to **350, 130**. Change the Text property of Form1 to **Many Buttons**.

3. Drag four Buttons onto the Form, and place them so that they are similar to the layout shown in Figure 13-27.

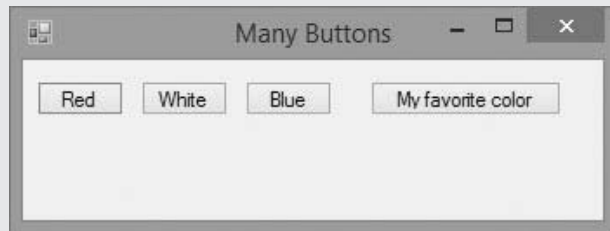


Figure 13-27 Four Buttons on the Many Buttons Form

- Change the Name properties of the buttons to `redButton`, `whiteButton`, `blueButton`, and `favoriteButton`. Change the Text on the Buttons to **Red**, **White**, **Blue**, and **My favorite color**, respectively. Adjust the size of the last button so its longer Text is fully displayed.
4. Examine the Properties list for the Red button. The `TabIndex` is 0. Examine the properties for the White, Blue, and My favorite color buttons. The IDE has set their `TabIndex` values to 1, 2, and 3, respectively.
 5. Click the **Save All** button, and then run the program. When the Form appears, the Red button has focus. Press the **Tab** key, and notice that focus changes to the White button. When you press **Tab** again, focus changes to the Blue button. Press **Tab** several more times, and observe that the focus rotates among the four Buttons.
 6. Dismiss the Form.
 7. Change the `TabIndex` property of the Blue button to **0**, and change the `TabIndex` of the Red button to **2**. (The `TabIndex` of the White button remains 1, and the `TabIndex` of the My favorite color button remains 3.)

(continues)

(continued)

Save the program again, and then run it. This time, the Blue button begins with focus. When you press Tab, the order in which the Buttons receive focus is Blue, then White, then Red, then My favorite color. (Clicking the Buttons or pressing Enter raises no event because you have not assigned events to the Buttons.)

8. Change the `TabIndex` property for the Red button back to **0** and the `TabIndex` property for the Blue button back to **2**. Click the **Save All** button.

Associating One Method with Multiple Events

In the next steps, you add three methods to the Many Buttons Form and cause one of the methods to execute each time the user clicks one of the four Buttons.

1. If it is not still open, open the **ManyButtons** project in the Visual Studio IDE.
2. Double-click the **Red** button on the Form to view the code for the shell of a `redButton_Click()` method. Between the method's curly braces, insert a statement that will change the Form's background color to red as follows:

```
this.BackColor = Color.Red;
```



At first glance, you might think `this` refers to the Button that is clicked. However, if you examine the `redButton_Click()` code in the `Form1.cs` file in the IDE, you will discover that the method is part of the `Form1` class. Therefore, `this.BackColor` refers to the Form's `BackColor` property.

3. Select the **Form1.cs [Design]** tab, and then double-click the **White** button. Add the following code to the `whiteButton_Click()` method that is generated:

```
this.BackColor = Color.White;
```

4. On the Form, double-click the **Blue** button. In its `Click()` method, add the following statement:

```
this.BackColor = Color.Blue;
```

5. On the form, click the **My favorite color** button. In its Properties list, click the **Events** button (the lightning bolt). Select the `Click` event. From the list box next to the `Click` event, select one of the three events to correspond to your favorite color of the three.

(continues)

(continued)



When you double-click a component on a Form in Design View, you move to the default method code. By clicking the component, however, you remain in Design View, and the correct set of properties appears in the Properties list.

647

6. Click the **Save All** button, and then execute the program. As you click **Buttons**, the Form's background color changes appropriately.
7. Dismiss the form, and exit Visual Studio.

Chapter Summary

- GUI programs are event driven—an event such as a button click “drives” the program to perform a task. Programmers also say a button click *raises an event*, *fires an event*, or *triggers an event*. A method that performs a task in response to an event is an event handler. The **Click** event is the event generated when a button is clicked.
- A delegate is an object that contains a reference to a method. **C#** delegates provide a way for a program to take alternative courses that are not predetermined; a delegate provides a way to pass a reference to a method as an argument to another method. You can assign one delegate to another using the **=** operator. You also can use the **+** and **+=** operators to combine delegates into a composed delegate that calls the delegates from which it is built.
- To declare your own event, you use a delegate. A client identifies methods to execute by associating the delegate with the method that should execute when the event occurs. The .NET Framework provides guidelines that the delegate type for an event should take exactly two parameters: a parameter indicating the source of the event, and an **EventArgs** parameter that encapsulates any additional information about the event. For events that do not use additional information, the .NET Framework has already defined an appropriate type named **EventHandler**.
- When you use existing **Control** components like **Buttons** and **ListBoxes**, they contain fields and **public** properties like **Text** as well as events with names like **Click**. A **Form** can contain any number of **Controls** that might have events associated with them. Additionally, a single control might be able to raise any number of events.
- Every mouse or keyboard event-handling method must have two parameters: an object representing the sender, and an object that holds event information. Mouse events include all the actions a user takes with a mouse, including clicking, pointing, and dragging. Mouse

events can be handled for any **Control** through an object of the class **MouseEventArgs**. The delegate used to create mouse event handlers is **MouseEventHandler**. Depending on the event, the type of the second parameter is **EventArgs** or **MouseEventArgs**. Keyboard events, also known as key events, occur when a user presses and releases keyboard keys. Depending on the event, the type of the second parameter is **EventArgs** or **KeyPressEventArgs**.

- When users encounter multiple GUI **Controls** on a **Form**, usually one **Control** has focus. That is, if the user presses Enter, the **Control** will raise an event. When a **Form** contains multiple **Controls**, you can create a separate event for each **Control**. However, you can also associate the same event with multiple **Controls**.
- When you encounter a new control in the IDE, you probably can use it without understanding all the code generated in the background. However, when you learn about the background code, your knowledge of **C#** is more complete.

Key Terms

An **event** is a reaction to an occurrence in a program.

"At runtime" describes actions that occur during a program's execution.

Event-driven programs contain code that causes an event such as a button click to drive the program to perform a task.

A button click **raises an event, fires an event, or triggers an event**.

An **event handler** is a method that performs a task in response to an event.

An **event receiver** is another name for an event handler.

An **event sender** is the control that generates an event.

A **click event** is an action fired when a user clicks a button in a GUI environment.

Event wiring is the act of connecting an event to its resulting actions.

A **delegate** is an object that contains a reference to a method.

A **composed delegate** calls the delegates from which it is built.

Invoking the event occurs when you call an event method.

Key events are keyboard events that occur when a user presses and releases keyboard keys.

Focus is the property of a **Control** that causes an event to be raised when the user presses Enter.

Review Questions

- In C#, events are _____ .
 - triggered by actions
 - handled by `catch` blocks
 - Boolean objects
 - only used in GUI programs
- A delegate is an object that contains a reference to a(n) _____ .
 - object
 - class
 - method
 - `Control`
- C# delegates provide a way for a program to _____ .
 - take alternative courses that are not determined until runtime
 - include multiple methods
 - include methods from other classes
 - include multiple `Controls` that use the same method
- Which of the following correctly declares a `delegate` type?
 - `void aDelegate(int num);`
 - `delegate void aDelegate(num);`
 - `delegate void aDelegate(int num);`
 - `delegate aDelegate(int num);`
- If you have declared a `delegate` instance, you can assign it a reference to a method as long as the method has the same _____ as the `delegate`.
 - return type
 - identifier
 - parameter list
 - two of the above
- You can combine two delegates to create a(n) _____ delegate.
 - assembled
 - classified
 - artificial
 - composed
- To combine two delegates using the `+` operator, the `delegate` objects must _____ .
 - have the same parameter list
 - have the same return type
 - both of these
 - neither of these

8. In C#, a(n) _____ is triggered when specific changes to an object occur.
- a. delegate
 - b. event
 - c. notification
 - d. instantiation
9. An event handler delegate requires _____ argument(s).
- a. zero
 - b. one
 - c. two
 - d. at least one
10. In an event-handler method, the sender is the _____.
- a. delegate associated with the event
 - b. method called by the event
 - c. object where the event was initiated
 - d. class containing the method that the event invokes
11. The `EventArgs` class contains a static field named _____.
- a. `Empty`
 - b. `Text`
 - c. `Location`
 - d. `Source`
12. When creating events, you can use a predefined delegate type named _____ that is automatically provided by the .NET Framework.
- a. `EventArgs`
 - b. `EventHandler`
 - c. `EventType`
 - d. `Event`
13. Which of the following is not a predefined `Control` event?
- a. `MouseEnter`
 - b. `Click`
 - c. `Destroy`
 - d. `TextChanged`
14. A single `Control` can raise _____ event(s).
- a. one
 - b. two
 - c. five
 - d. any number of
15. When you create `Forms` with `Controls` that raise events, an advantage to creating the code by hand over using the Visual Studio IDE is _____.
- a. you are less likely to make typing errors
 - b. you save a lot of repetitious typing
 - c. you are less likely to forget to set a property
 - d. you gain a clearer understanding of the C# language

16. When a **Form** contains three **Control**s and one has focus, you can raise an event by _____.
- a. clicking any **Control**
 - b. pressing Enter
 - c. either of these
 - d. none of these
17. The **TabStop** property of a **Control** is a(n) _____.
- a. integer value indicating the tab order
 - b. Boolean value indicating whether the **Control** has a position in the tab sequence
 - c. string value indicating the name of the method executed when the **Control** raises an event
 - d. **delegate** name indicating the event raised when the user tabs to the **Control**
18. The **TabIndex** property of a **Control** is a(n) _____.
- a. integer value indicating the tab order
 - b. Boolean value indicating whether the **Control** has a position in the tab sequence
 - c. string value indicating the name of the method executed when the **Control** raises an event
 - d. **delegate** name indicating the event raised when the user tabs to the **Control**
19. The **Control** that causes an event is the _____ argument to an event method.
- a. first
 - b. second
 - c. third
 - d. fourth
20. Which of the following is true?
- a. A single event can be generated from multiple **Control**s.
 - b. Multiple events can be generated from a single **Control**.
 - c. Both of the above are true.
 - d. None of the above are true.

Exercises



Programming Exercises

652

1. Create a project named **WordsOfWisdom** with a **Form** containing at least four **Label**s that hold “wise” quotes of your choice. When the program starts, the background color of the **Form** and each **Label** should be black. When the user passes a mouse over a **Label**, change its **BackColor** to white, revealing the text of the quote.
2. Create a project named **RecentlyVisitedSites** that contains a **Form** with a list of three **LinkLabel**s that link to any three Web sites you choose. When a user clicks a **LinkLabel**, link to that site. When a user’s mouse hovers over a **LinkLabel**, display a brief message that explains the site’s purpose. After a user clicks a link, move the most recently selected link to the top of the list, and move the other two links down, making sure to retain the correct explanation with each link.
3. Create a project named **ClassicBookSelector** that contains a **Form** with a **ListBox** that lists at least five classic books that you think all educated people should have read. When the user places the mouse over the **ListBox**, display a **Label** that contains a general statement about the benefits of reading. The **Label** disappears when the user’s mouse leaves the **ListBox** area. When the user clicks a book title in the **ListBox**, display another **Label** that contains a brief synopsis of the specific book. Also change the **BackColor** of the **Form** to a different color for each book.
4. Locate an animated .gif file on the Web. Create a project named **Animated** with a **Form** that contains a **PictureBox**. Display three different messages on a **Label**—one when the user’s mouse is over the **PictureBox**, one when the mouse is not over the **PictureBox**, and one when the user clicks the **PictureBox**.
5. Create a project named **FlorencesFloralCreations** that allows a user to use a **ListBox** to choose an occasion for which to send a floral creation (for example *Get well*). When the user selects an occasion, the program should display a second **ListBox** that contains at least two main flower choices for each occasion type—for example, daisy or rose. After the user selects a flower type, the program should display a congratulatory message on a **Label** indicating that the choice is a good one, and the flower choice **ListBox** also becomes invisible. If the user makes a new selection from the occasion **ListBox**, the congratulatory message is invisible until the user selects a new flower option.

Hint: You can remove the entire contents of a **ListBox** using the `Items.Clear()` method, as in `this.listBox1.Items.Clear();`

6. Create a project named **GuessANumber** with a **Form** that contains a guessing game with five **RadioButton**s numbered 1 through 5. Randomly choose one of the **RadioButton**s as the winning button. When the user clicks a **RadioButton**, display a message indicating whether the user is right.

Add a **Label** to the **Form** that provides a hint. When the user's mouse hovers over the label, notify the user of one **RadioButton** that is incorrect. After the user makes a selection, disable all the **RadioButton**s.



You can create a random number that is at least `min` but less than `max` using the following statements:

```
Random ran = new Random();
int randomNumber;
randomNumber = ran.Next(min, max);
```

7. Create a project named **PickLarger** with a **Form** that contains two randomly generated arrays, each containing 100 numbers. Include two **Buttons** labeled "1" and "2". Starting with position 0 in each array, ask the user to guess which of the two arrays contains the higher number and to click one of the two buttons to indicate the guess. After each button click, the program displays the values of the two compared numbers, as well as running counts of the number of correct and incorrect guesses. After the user makes a guess, disable the **Buttons** while the user views the results. After clicking a **Next Button**, the user can make another guess using the next two array values. If the user makes more than 100 guesses, the program should reset the array subscript to 0 so the comparisons start over but continue to keep a running score.



Debugging Exercises

1. Each of the following files or projects in the `Chapter.13` folder of your downloadable student files has syntax and/or logical errors. Immediately save the two project folders with their new names before starting to correct their errors. After you correct the errors, save each file or project using the same filename preceded with *Fixed*. For example, the file `DebugThirteen1.cs` will become `FixedDebugThirteen1.cs` and the project folder for `DebugThirteen3` will become `FixedDebugThirteen3`.
 - a. `DebugThirteen1.cs`
 - b. `DebugThirteen2.cs`
 - c. `DebugThirteen3`
 - d. `DebugThirteen4`



Case Problems

1. In Chapter 12, you created an interactive advertisement named **GreenvilleAdvertisement** that can be used to recruit contestants for the Greenville Idol competition. Now, modify that program to include at least one handled event using the techniques you learned in this chapter.
2. In Chapter 12, you created an interactive advertisement named **MarshallsAdvertisement** that can be used to advertise the available murals painted by Marshall's Murals. Now, modify that program to include at least one handled event using the techniques you learned in this chapter.