Chapter 13

Voigtlander Bessa-L / 15mm Super Wide-Heliar

Fairview Park

Custom Events

Learning Objectives

- *List and describe the components required to implement custom events*
- *Describe the purpose and use of delegates*
- *Describe the purpose and use of events*
- *Use the "delegate" keyword to declare new delegate types*
- *Use the "event" keyword to declare new event members*
- *Create a custom event argument passing class*
- *Create event-argument objects and pass to event handler methods*
- *Describe the event processing cycle*
- *Create and use event publisher classes*
- *Create and use event subscriber classes*

Introduction

Graphical User Interface (GUI) components are not the only type of objects that can have event members. Indeed, you can add events to the classes you design, and that's the subject of this chapter.

To add custom events to your programs, you'll need to know a little something about the following topics: 1) how to use the *delegate* keyword to declare new delegate types, 2) how to use the *event* keyword to declare new event members using delegate types, 3) how to create a class that conveys event data between an event publisher and an event subscriber, 4) how to create an event publisher, 5) how to create an event subscriber, 6) how to create event handler methods, and 7) how to register event handlers with a particular event. If you have read Chapter 12, you already know how to do items 6 and 7 on the list.

The information and programming techniques you learn from reading this chapter will open up a whole new world of programming possibilities.

C# Event Processing Model: An Overview

C# is a modern programming language supported by an extensive Application Programming Interface (API) referred to as the .NET Framework. And, as you learned in Chapter 12, C# also supports event-driven programming normally associated with Microsoft Windows applications. One normally thinks of events as being generated exclusively by GUI components, but any object can generate an event if it's programmed to do so.

You need two logical components to implement the event processing model: 1) an event producer (or *publisher*), and 2) an event consumer (or *subscriber*). Each component has certain responsibilities. Consider the following diagram:

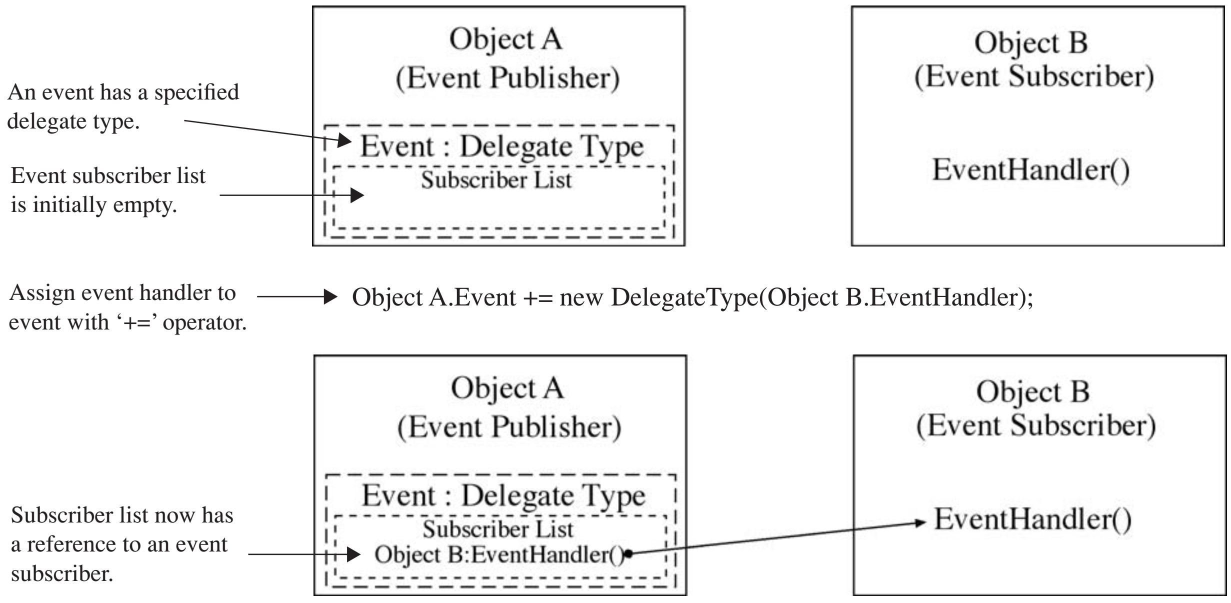

Object A
(Event Publisher)
Object B
(Event Subscriber)
An event has a specified delegate type.
Event : Delegate Type
Subscriber List
Event subscriber list is initially empty.
EventHandler()
Assign event handler to event with '+=' operator.
Object A.Event += new DelegateType(Object B.EventHandler);
Object A
(Event Publisher)
Object B
(Event Subscriber)
Event : Delegate Type
Subscriber List
Object B:EventHandler()
Subscriber list now has a reference to an event subscriber.
EventHandler()

Figure 13-1: Event Publisher and Subscriber

Referring to Figure 13-1 — each event in Object A has a specified delegate type. The delegate type specifies the authorized method signature for event handler methods. However, the delegate does more than just specify a method signature; a delegate object maintains a list of event subscribers in the form of references to event handler methods. These references can point to an object and one of its instance methods or to a static method. The event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. The EventHandler() method defined in Object B must conform to the method signature specified by the event's delegate type. If you attempt to use an event handler method that does not conform to the delegate type's method signature, you will receive a compiler error. Let's now substitute some familiar names for Object A and Object B. Figure 13-2 offers a revised diagram.

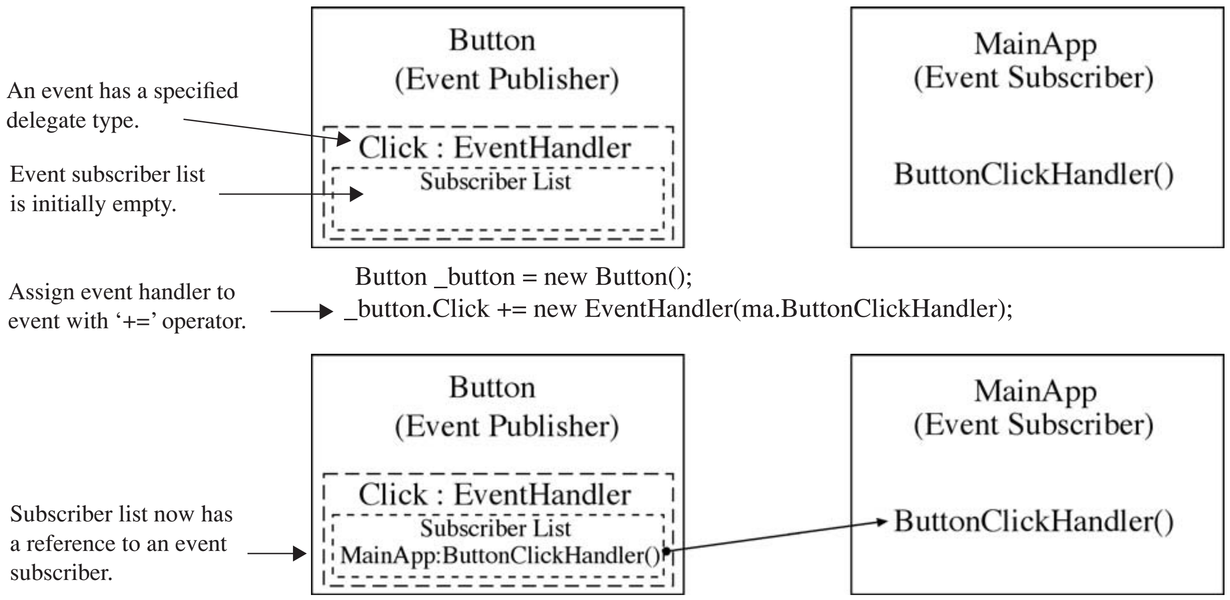

An event has a specified delegate type.
Event subscriber list is initially empty.
Button (Event Publisher)
Click : EventHandler
Subscriber List
MainApp (Event Subscriber)
ButtonClickHandler()
Assign event handler to event with '+=' operator.
Button _button = new Button();
_button.Click += new EventHandler(ma.ButtonClickHandler);
Button (Event Publisher)
Click : EventHandler
Subscriber List
MainApp:ButtonClickHandler()
MainApp (Event Subscriber)
ButtonClickHandler()
Subscriber list now has a reference to an event subscriber.

Figure 13-2: Event Publisher and Subscriber

Referring to Figure 13-2 — a button's Click event has an EventHandler delegate type. As you learned in Chapter 12, event handler methods assigned to a button's Click event must have the following signature:

```
void MethodName(object sender, EventArgs e)
```

The method's name can be pretty much anything you want it to be, but it must declare two parameters, the first of type *object*, and the second of type *EventArgs*, and it must return void. The names of the parameters can be anything you want as well, but the names *sender* and *e* work just fine.

When the button's Click event is fired, the Click event's delegate instance calls each registered subscriber's event handler method in the order it appears in the event subscriber list. This is all handled behind the scenes as you will soon see. In the case of a button and other controls, there exists an internal method that is called when a Click event occurs that kicks off the subscriber notification process. Remember that when talking about GUI components, a mouse click results in the generation of a message that is ultimately routed to the window in which the mouse click occurred. This message is translated into a Click event. When writing custom events, you can intercept messages and translate them into events or write a method that generates events out of thin air or in response to some other stimulus.

Quick Review

You need two logical components to implement the event processing model: 1) an event producer (*publisher*), and 2) an event consumer (*subscriber*). A *delegate* type specifies the authorized method signature for event handler methods. A delegate object maintains a list of event subscribers in the form of references to event handler methods. An event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. Event handler methods must conform to the method signature specified by an event's delegate type.

Custom Events Example: Minute Tick

The best way to get your head around custom events is to study an example application. This section presents a short program that implements custom events. The Minute Tick application consists of five source files, four of which appear in the Unified Modeling Language (UML) class diagram shown in Figure 13-3. Referring to Figure 13-3 — both the Publisher and Subscriber classes depend on the MinuteEventArgs class and the ElapsedMinuteEventHandler delegate. The Publisher class contains a MinuteTick event, which is of type ElapsedMinuteEventHandler. The

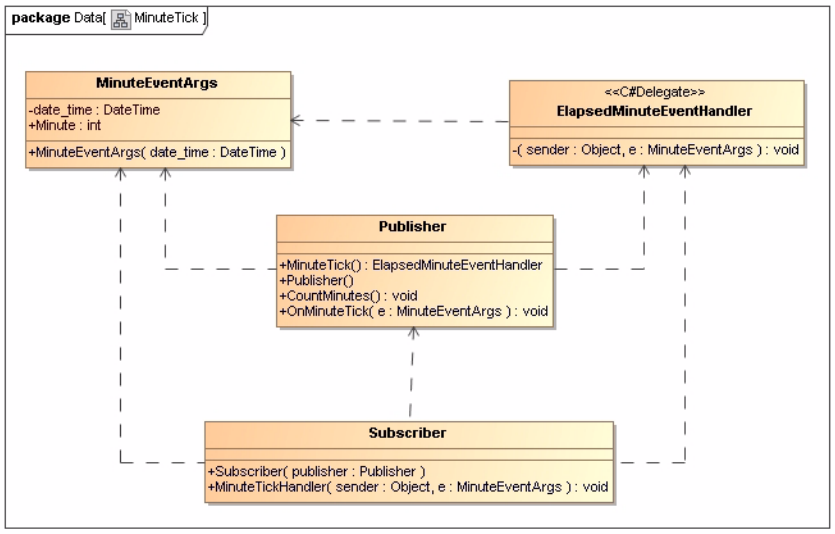

package Data[MinuteTick]
MinuteEventArgs
-date_time : DateTime
+Minute : int
+MinuteEventArgs(date_time : DateTime)
<<C#Delegate>>
ElapsedMinuteEventHandler
-(sender : Object, e : MinuteEventArgs) : void
Publisher
+MinuteTick() : ElapsedMinuteEventHandler
+Publisher()
+CountMinutes() : void
+OnMinuteTick(e : MinuteEventArgs) : void
Subscriber
+Subscriber(publisher : Publisher)
+MinuteTickHandler(sender : Object, e : MinuteEventArgs) : void

Figure 13-3: Minute Tick UML Class Diagram

ElapsedMinuteEventHandler delegate depends on the MinuteEventArgs class because it is the type of one of its parameters, as is shown in the diagram.

The complete Minute Tick application source code is given in Examples 13.1 through 13.5.

13.1 MinuteEventArgs.cs

```
using System;

public class MinuteEventArgs : EventArgs {
  private DateTime date_time;

  public MinuteEventArgs(DateTime date_time){
    this.date_time = date_time;
  }

  public int Minute {
    get {  return date_time.Minute; }
  }
}
```

Referring to Example 13.1 — the MinuteEventArgs class extends the EventArgs class and adds a private field named *date_time* and one public read-only property named *Minute,* which simply returns the value of the Minute property of the DateTime object.

13.2 ElapsedMinuteEventHandler.cs

```
using System;

public delegate void ElapsedMinuteEventHandler(Object sender, MinuteEventArgs e);
```

Referring to Example 13.2 — the ElapsedMinuteEventHandler delegate specifies a method signature that returns `void` and takes two parameters, the first one an object, and the second one of type MinuteEventArgs.

13.3 Publisher.cs

```
using System;

public class Publisher {

   public event ElapsedMinuteEventHandler MinuteTick;


   public Publisher(){
     Console.WriteLine("Publisher Created");
   }

   public void CountMinutes(){
     int current_minute = DateTime.Now.Minute;
     while(true){
       if(current_minute != DateTime.Now.Minute){
          Console.WriteLine("Publisher: {0}", DateTime.Now.Minute);
          OnMinuteTick(new MinuteEventArgs(DateTime.Now));
          current_minute = DateTime.Now.Minute;
19          }//end if
20        }  // end while
21      }  // end CountMinutes method
22
23      public void OnMinuteTick(MinuteEventArgs e){
24         if(MinuteTick != null){
25            MinuteTick(this, e);
26          }
27      }// end OnMinuteTick method
28   }  // end Publisher class definition
```

Referring to Example 13.3 — the Publisher class defines an event named *MinuteTick*. Notice that the MinuteTick event is of type ElapsedMinuteEventHandler. The CountMinutes() method that starts on line 12 contains a `while` loop that repeats forever and continuously compares the values of the current_minute with DateTime.Now.Minute. As soon as a change is detected in the two values, a brief message is written to the console followed by a call to the publisher's OnMinuteTick() method on line 17. Notice that when this method is called, a new MinuteEventArgs object is created and used as an argument to the method call. The OnMinuteTick() method definition begins on line 23. It takes the MinuteEventArgs parameter and passes it on to a call to the MinuteTick event. Note on line 24 how the `if` statement checks to see if the MinuteTick reference is null. It will be null if no event handler methods have been registered with the event.

13.4 Subscriber.cs

```
1    using System;
2
3    public class Subscriber {
4
5      public Subscriber(Publisher publisher){
6       publisher.MinuteTick += new ElapsedMinuteEventHandler(this.MinuteTickHandler);
7        Console.WriteLine("Subscriber Created");
8      }
9
10     public void MinuteTickHandler(Object sender, MinuteEventArgs e){
11       Console.WriteLine("Subscriber Handler Method: {0}", e.Minute);
12     }
13    } // end Subscriber class definition
```

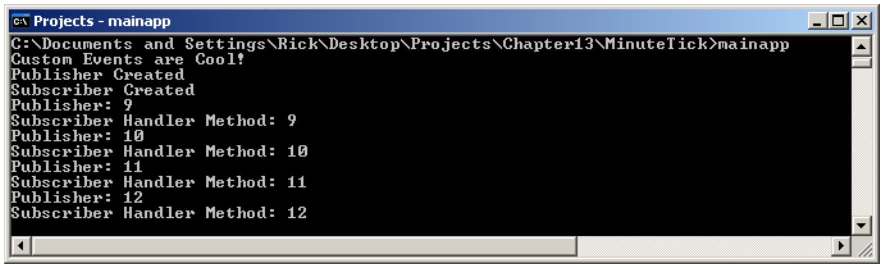
Referring to Example 13.4 — the Subscriber class declares an event handler method on line 10 named *Minute-TickHandler()*. The MinuteTickHandler() method defines two arguments of the types required by the ElapsedMinute-EventHandler delegate type. The ElapsedMinuteEventHandler delegate is used on line 6 to register the subscriber's MinuteTickHandler() method with the publisher's MinuteTick event.

13.5 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4     public static void Main(){
5       Console.WriteLine("Custom Events are Cool!");
6
7       Publisher p = new Publisher();
8       Subscriber s = new Subscriber(p);
9       p.CountMinutes();
10
11    }  // end main
12   }  //end MainApp class definition
```

Referring to Example 13.5 — the MainApp class provides the Main() method. It simply creates a Publisher object and a Subscriber object, and then makes a call to the publisher's CountMinutes() method. Figure 13-4 shows the results of running this application. Note that the actual minutes displayed when the program runs depend on when you start the program.

Application started at 9 minutes past the hour. Your time outputs will reflect the time you run the program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter13\MinuteTick>mainapp
Custom Events are Cool!
Publisher Created
Subscriber Created
Publisher: 9
Subscriber Handler Method: 9
Publisher: 10
Subscriber Handler Method: 10
Publisher: 11
Subscriber Handler Method: 11
Publisher: 12
Subscriber Handler Method: 12
```

Figure 13-4: Results of Running Example 13.5

Custom Events Example: Automated Water Tank System

In this section, I want to show you how you might model a complex system using custom events. The system modeled here is a simple water tank that can be filled with water. Once the water reaches a certain level within the tank, a pump is activated and drains the tank until it again reaches a certain level. The tank contains two water level sensors. One acts as a high-level sensor and the other acts as a low-level sensor. The tank also has a pump that pumps water at a certain rate or pumping capacity. The system comprises the following classes: *Pump*, *WaterLevelEventArgs*, *WaterLevelEventHandler*, *WaterTank*, and *WaterSystemApp,* which serves as the main application class. Figure 13-5 gives the UML class diagram for the water tank system.

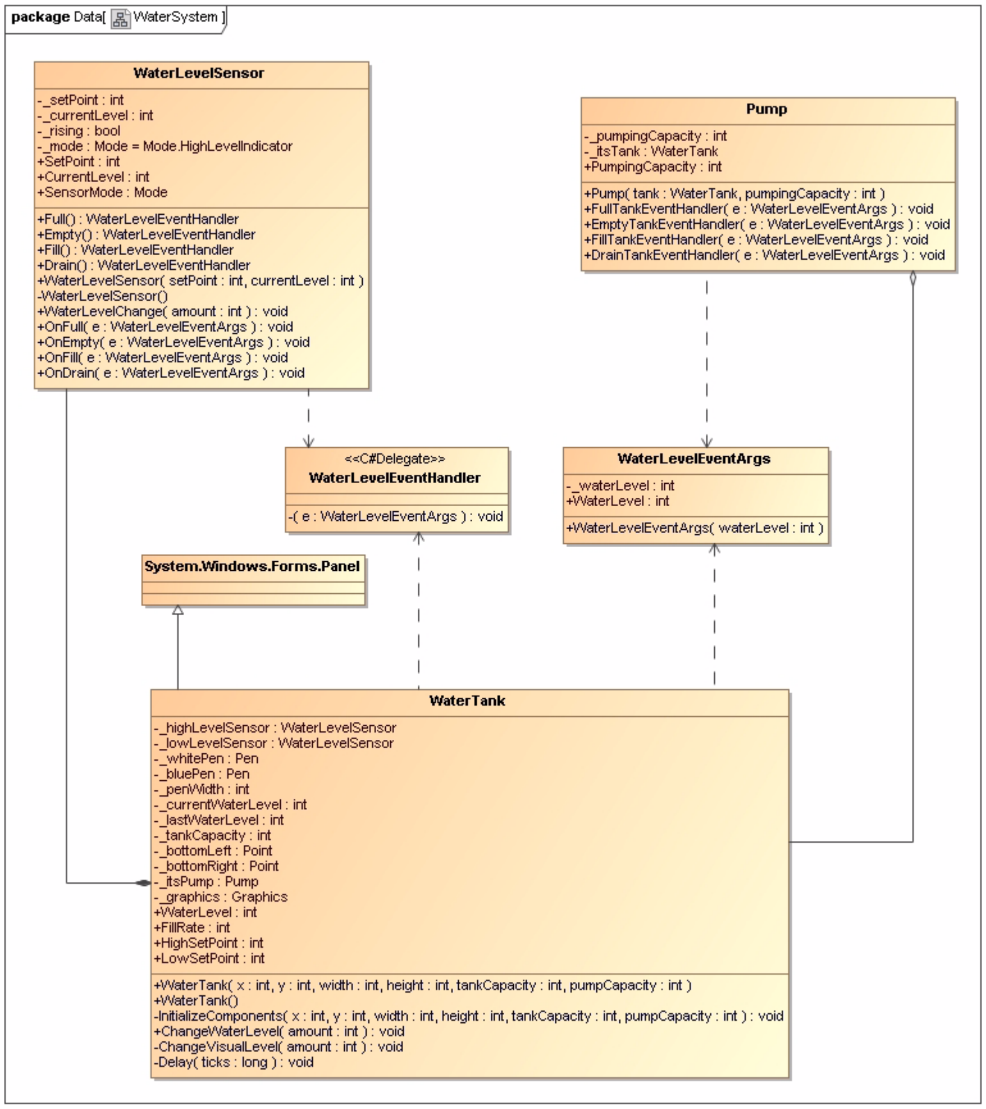

package Data[WaterSystem]
WaterLevelSensor
-_setPoint : int
-_currentLevel : int
-_rising : bool
-_mode : Mode = Mode.HighLevelIndicator
+SetPoint : int
+CurrentLevel : int
+SensorMode : Mode
+Full() : WaterLevelEventHandler
+Empty() : WaterLevelEventHandler
+Fill() : WaterLevelEventHandler
+Drain() : WaterLevelEventHandler
+WaterLevelSensor(setPoint : int, currentLevel : int)
-WaterLevelSensor()
+WaterLevelChange(amount : int) : void
+OnFull(e : WaterLevelEventArgs) : void
+OnEmpty(e : WaterLevelEventArgs) : void
+OnFill(e : WaterLevelEventArgs) : void
+OnDrain(e : WaterLevelEventArgs) : void
Pump
-_pumpingCapacity : int
-_itsTank : WaterTank
+PumpingCapacity : int
+Pump(tank : WaterTank, pumpingCapacity : int)
+FullTankEventHandler(e : WaterLevelEventArgs) : void
+EmptyTankEventHandler(e : WaterLevelEventArgs) : void
+FillTankEventHandler(e : WaterLevelEventArgs) : void
+DrainTankEventHandler(e : WaterLevelEventArgs) : void
<<C#Delegate>>
WaterLevelEventHandler
-(e : WaterLevelEventArgs) : void
WaterLevelEventArgs
-_waterLevel : int
+WaterLevel : int
+WaterLevelEventArgs(waterLevel : int)
System.Windows.Forms.Panel
WaterTank
-_highLevelSensor : WaterLevelSensor
-_lowLevelSensor : WaterLevelSensor
-_whitePen : Pen
-_bluePen : Pen
-_penWidth : int
-_currentWaterLevel : int
-_lastWaterLevel : int
-_tankCapacity : int
-_bottomLeft : Point
-_bottomRight : Point
-_itsPump : Pump
-_graphics : Graphics
+WaterLevel : int
+FillRate : int
+HighSetPoint : int
+LowSetPoint : int
+WaterTank(x : int, y : int, width : int, height : int, tankCapacity : int, pumpCapacity : int)
+WaterTank()
-InitializeComponents(x : int, y : int, width : int, height : int, tankCapacity : int, pumpCapacity : int) : void
+ChangeWaterLevel(amount : int) : void
-ChangeVisualLevel(amount : int) : void
-Delay(ticks : long) : void

Figure 13-5: Water Tank System UML Class Diagram

Referring to Figure 13-5 — The WaterTank class extends the System.Windows.Forms.Panel class. This gives the water tank a visual representation. When water is added to the tank, the panel is filled in with blue lines as the water level rises. The lines are then overdrawn with a different color as the water level recedes. The WaterLevelEventArgs class is used to pass water level information between event publisher and subscriber. In this example, the WaterLevelSensor is the publisher and the Pump is the subscriber. The WaterLevelEventHandler delegate is used to declare the WaterLevelSensor's Fill, Full, Drain, and Empty events. Let's take a look at the code.

13.6 WaterLevelEventArgs.cs

```
using System;

public class WaterLevelEventArgs : EventArgs {

  private int _waterLevel;

  public WaterLevelEventArgs(int waterLevel){
     WaterLevel = waterLevel;
  }

  public int WaterLevel {
     get { return _waterLevel; }
     set { _waterLevel = value; }
  }
}
```

Referring to Example 13.6 — the WaterLevelEventArgs class contains one private integer field named *_waterLevel* and one public property named *WaterLevel*.

13.7 WaterLevelEventHandler.cs

```
using System;

public delegate void WaterLevelEventHandler(WaterLevelEventArgs e);
```

Referring to Example 13.7 — the WaterLevelEventHandler delegate specifies an event-handler method signature that returns `void` and contains one parameter of type WaterLevelEventArgs.

13.8 WaterLevelSensor.cs

```
using System;

public class WaterLevelSensor {
   private int _setPoint;
   private int _currentLevel;
   private bool _rising;
   private Mode _mode = Mode.HighLevelIndicator;


   public enum Mode { HighLevelIndicator, LowLevelIndicator };
   public event WaterLevelEventHandler Full;
   public event WaterLevelEventHandler Empty;
   public event WaterLevelEventHandler Fill;
   public event WaterLevelEventHandler Drain;

   public int SetPoint {
      get { return _setPoint; }
      set { _setPoint = value; }
   }

   public int CurrentLevel {
      get { return _currentLevel; }
      set { _currentLevel = value; }
   }

   public Mode SensorMode {
      get { return _mode; }
      set { _mode = value; }
   }

   public WaterLevelSensor(int setPoint, int currentLevel){
      SetPoint = setPoint;
      CurrentLevel = currentLevel;
   }

   private WaterLevelSensor(){ }

   public void WaterLevelChange(int amount){
      int lastLevel = CurrentLevel;
      CurrentLevel += amount;
      _rising = (CurrentLevel >= lastLevel);

      switch(_mode){
         case Mode.HighLevelIndicator :
               if(_rising){
                 if(CurrentLevel >= SetPoint){
                      WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
                      OnFull(args);
                   } else{
                      WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
                      OnFill(args);
                   }
               }
               break;

          case Mode.LowLevelIndicator :
            if(!_rising){
            if(CurrentLevel <= SetPoint){
                WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
                OnEmpty(args);
              } else{
                 WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
                 OnDrain(args);
              }
           }
           break;
      } // end switch
  }

    public void OnFull(WaterLevelEventArgs e){
      if(Full != null){
         Full(e);
      }
     }

    public void OnEmpty(WaterLevelEventArgs e){
       if(Empty != null){
         Empty(e);
       }
    }

    public void OnFill(WaterLevelEventArgs e){
      if(Fill != null){
         Fill(e);
       }
    }

    public void OnDrain(WaterLevelEventArgs e){
      if(Drain != null){
         Drain(e);
      }
    }
 }// end WaterLevelClass definition
```

Referring to Example 13.8 — the WaterLevelSensor is a primary component of the water system. Essentially, its purpose is to keep track of a tank's water level. A WaterLevelSensor object functions in one of two modes of operation as defined by the *Mode* enumeration. It can be either a *HighLevelIndicator* or a *LowLevelIndicator*. If it's operating as a HighLevelIndicator, it keeps track of rising water added via the WaterLevelChange() method. If the water level is rising, it fires the *Fill* event. When the water level reaches the set point, it fires the *Full* event.

If a WaterLevelSensor is operating in the LowLevelIndicator mode, it responds to falling water levels by firing the *Drain* event until the water reaches the low set point, at which time it fires the *Empty* event.

Each of the four events — Fill, Full, Drain, and Empty — are of type WaterLevelEventHander delegate. The Pump class, shown in the following example, defines four event handler methods that respond to each of these events.

13.9 Pump.cs

```
 using System;

 public class Pump {

    private int _pumpingCapacity;
    private WaterTank _itsTank;

    public int PumpingCapacity {
       get {  return _pumpingCapacity; }
       set {  _pumpingCapacity = value; }
    }

    public Pump(WaterTank tank, int pumpingCapacity){
       PumpingCapacity = pumpingCapacity;
       _itsTank = tank;
    }

    public void FullTankEventHandler(WaterLevelEventArgs e){
       Console.WriteLine("FullTankEventHandler: Draining the water tank!");
       _itsTank.ChangeWaterLevel(-PumpingCapacity);
    }

    public void EmptyTankEventHandler(WaterLevelEventArgs e){
       Console.Write("EmptyTankEventHandler: ");
```

```
        Console.WriteLine("Water tank has been drained! The water tank contains " +
                          e.WaterLevel + " gallons!");
     }

     public void FillTankEventHandler(WaterLevelEventArgs e){
        Console.Write("FillTankEventHandler: ");
        Console.WriteLine("The water tank contains " + e.WaterLevel + " gallons!");
     }

     public void DrainTankEventHandler(WaterLevelEventArgs e){
        Console.Write("DrainTankEventHandler: ");
        Console.WriteLine("The water tank contains " + e.WaterLevel + " gallons!");
        _itsTank.ChangeWaterLevel(-PumpingCapacity);

     }

  }
```

Referring to Example 13.9 — a Pump object is created with an associated WaterTank object and a pumping capacity. Water added to the tank causes a Fill event to fire. The FillTankEventHandler() method responds by printing the value of the tank's current water level to the console. Note that the current water level is determined by reading the WaterLevelEventArgs.WaterLevel property. When the water level reaches the high level sensor's set point, the sensor fires the Full event that calls the Pump's FullTankEventHandler() method. This starts the automatic draining process by calling the WaterTank.ChangeWaterLevel() method with a negative amount of water equal to the volume of its pumping capacity. This in turn triggers a Drain event in a WaterLevelSensor object, which calls the pump's DrainTankEventHandler() method. This results in yet another call (recursive) to the WaterTank.ChangeWaterLevel() method with a negative amount of water equal to the volume of its pumping capacity. Thus, the recursive calls to the DrainTankEventHandler() method repeat until the low-level indicator reaches its set point.

13.10 WaterTank.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class WaterTank : Panel {

   // Private fields
   private WaterLevelSensor _highLevelSensor;
   private WaterLevelSensor _lowLevelSensor;
   private Pen _whitePen;
   private Pen _bluePen;
   private int _penWidth;
   private int _currentWaterLevel;
   private int _lastWaterLevel;
   private int _tankCapacity;
   private Point _bottomLeft;
   private Point _bottomRight;
   private Pump _itsPump;
   private Graphics _graphics;

  // Constants
   private const int UPPER_LEFT_CORNER_X = 100;
   private const int UPPER_LEFT_CORNER_Y = 100;
   private const int WIDTH = 100;
   private const int HEIGHT = 500;
   private const int TANK_CAPACITY = 10000;
   private const int PUMP_CAPACITY = 1000;
   private const int ONE_PIXEL_WIDE = 1;
   private const int EMPTY = 0;

  public int WaterLevel {
     get { return _currentWaterLevel; }
  }

   public int FillRate {
     get { return _itsPump.PumpingCapacity; }
  }

   public int HighSetPoint {
     get {  return _highLevelSensor.SetPoint; }
     set {  _highLevelSensor.SetPoint = value; }
  }

   public int LowSetPoint {
     get {  return _lowLevelSensor.SetPoint; }
     set {  _lowLevelSensor.SetPoint = value; }
  }
```

```

    public WaterTank(int x, int y, int width, int height, int tankCapacity, int pumpCapacity){
       this.InitializeComponents(x, y, width, height, tankCapacity, pumpCapacity);
    }

    public WaterTank():this(UPPER_LEFT_CORNER_X, UPPER_LEFT_CORNER_Y, WIDTH, HEIGHT, TANK_CAPACITY,
                           PUMP_CAPACITY){ }

    private void InitializeComponents(int x, int y, int width, int height, int tankCapacity,
                                      int pumpCapacity){

       this.Bounds = new Rectangle(x, y, width, height);
       this.BackColor = Color.White;
       this.BorderStyle = BorderStyle.Fixed3D;
       _graphics = this.CreateGraphics();
       _bottomLeft = new Point(0, height);
       _bottomRight = new Point(width, height);
       _tankCapacity = tankCapacity;
       _currentWaterLevel = EMPTY;
       _itsPump = new Pump(this, pumpCapacity);
       _penWidth = this.Height/(_tankCapacity/_itsPump.PumpingCapacity);
       if(_penWidth < 1) _penWidth = 1;
       _whitePen = new Pen(Color.White, _penWidth);
       _bluePen = new Pen(Color.Blue, _penWidth);
       _highLevelSensor = new WaterLevelSensor(tankCapacity - pumpCapacity, EMPTY);
       _highLevelSensor.SensorMode = WaterLevelSensor.Mode.HighLevelIndicator;
       _highLevelSensor.Fill += new WaterLevelEventHandler(_itsPump.FillTankEventHandler);
       _highLevelSensor.Full += new WaterLevelEventHandler(_itsPump.FullTankEventHandler);
       _lowLevelSensor = new WaterLevelSensor(pumpCapacity, EMPTY);
       _lowLevelSensor.SensorMode = WaterLevelSensor.Mode.LowLevelIndicator;
       _lowLevelSensor.Drain += new WaterLevelEventHandler(_itsPump.DrainTankEventHandler);
       _lowLevelSensor.Empty += new WaterLevelEventHandler(_itsPump.EmptyTankEventHandler);
    }

    public void ChangeWaterLevel(int amount){
       _lowLevelSensor.WaterLevelChange(amount);
       _highLevelSensor.WaterLevelChange(amount);
       _currentWaterLevel += amount;
       _lastWaterLevel = _currentWaterLevel;
       this.ChangeVisualLevel(amount);
    }

    private void ChangeVisualLevel(int amount){
       if(amount > 0){
          _graphics.DrawLine(_bluePen, _bottomLeft, _bottomRight);
          _bottomLeft.Y -= _penWidth;
          _bottomRight.Y -= _penWidth;

       }else{
          _graphics.DrawLine(_whitePen, _bottomLeft, _bottomRight);
          _bottomLeft.Y += _penWidth;
          _bottomRight.Y += _penWidth;
          Delay(30000000);
       }

    } // end ChangeVisualLevel method

    private void Delay(long ticks){
       for(long i = 0; i<ticks; i++){
          ;
       }
    }

 } // end class definition

```

Referring to Example 13.10 — the WaterTank class is an aggregate of a Pump and two WaterLevelSensors. It also provides a visual representation of a water tank by animating the rising and falling water level via blue and white lines drawn on a Panel. Most of the action occurs in three methods: InitializeComponents(), ChangeWaterLevel(), and ChangeVisualLevel(). (**Note:** An attempt is made to keep the visual filling animation in step with the tank's water level, however, when the value of _penWidth reaches 1, the animation gets a little goofy!)

The WaterLevelSensor objects are created in the InitializeComponents() method. One is designated as the _highLevelSensor and the other the _lowLevelSensor. Each sensor's SetPoint is set via its constructor followed by its SensorMode property. Next, the Pump's event handler methods are registered with each sensor's respective events.

Water is added to the tank via the ChangeWaterLevel() method. This in turn makes a call to each sensor's Water-LevelChange() method. The tank's level values are adjusted and finally its visual state is changed with a call to its

ChangeVisualLevel() method. The Delay() method is used to slow down the draining animation so you can watch the water level drop.

13.11 WaterSystemApp.cs

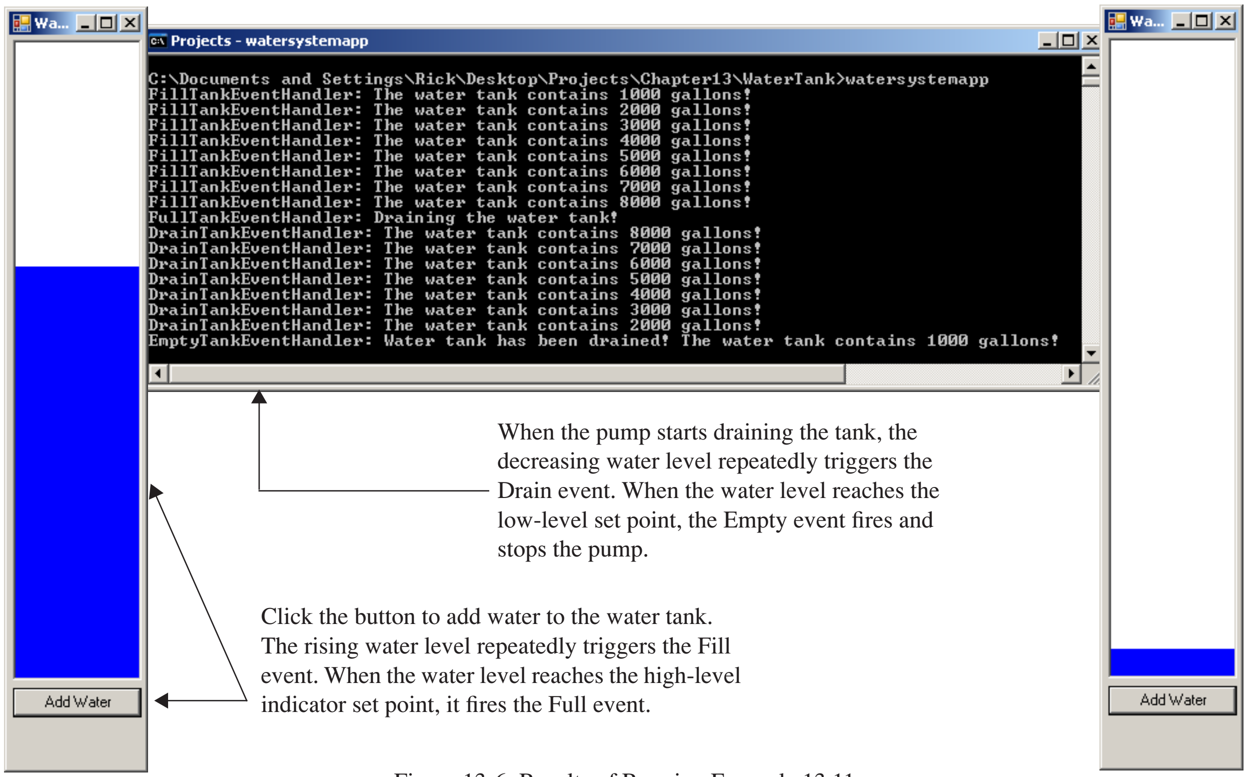
```
using System;
using System.Windows.Forms;
using System.Drawing;

public class WaterSystemApp : Form {

  private FlowLayoutPanel _panel;
  private Button _button;
  private WaterTank _tank;

  public WaterSystemApp(){
    this.InitializeComponents();
  }

  public void InitializeComponents(){
    _tank = new WaterTank();
    _button = new Button();
    _button.Text = "Add Water";
    _button.Click += new EventHandler(this.AddWaterButtonClick);
    _button.Dock = DockStyle.Bottom;
    _panel = new FlowLayoutPanel();
    _panel.SuspendLayout();
    _panel.FlowDirection = FlowDirection.TopDown;
    _panel.AutoSize = true;
    _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
    _panel.Height = _tank.Height + _button.Height + 75;
    _panel.Controls.Add(_tank);
    _panel.Controls.Add(_button);
    this.SuspendLayout();
    this.Text = "Water System";
    this.Height = _panel.Height;
    this.Width = _tank.Width;
    this.Controls.Add(_panel);
    _panel.ResumeLayout();
    this.ResumeLayout();
  }

  public void AddWaterButtonClick(object sender, EventArgs e){
     _tank.ChangeWaterLevel(_tank.FillRate);
  }

  public static void Main(){
    Application.Run(new WaterSystemApp());
  }

}// end WaterSystemApp class definition
```

Referring to Example 13.11 — the WaterSystemApp class extends Form and provides the user interface for the water system application. It creates a FlowLayoutPanel and adds to it the WaterTank, which is itself a panel, and a button. Each time the button is clicked, water is added to the tank in an amount equal to the tank's *FillRate* property. The WaterTank.FillRate property is read-only and equals the value of its pump's PumpingCapacity. Figure 13-6 shows the results of running this program. However, you'll learn more from the program by running it and seeing for yourself how the events actually work. Experimenting with different tank dimensions and pumping capacities is left as an exercise.

Naming Conventions

If you'll pause for a moment to consider the previous two custom event examples, you'll notice a few similarities in the names given to certain components and methods. It helps to clarify the purpose of each component or method by adopting the following or similar naming convention.

- Add the suffix "EventArgs" to your event argument class names.
- Add the suffix "EventHandler" to your event-handler delegate names.
- Add the prefix "On" to the event name for the method that fires the event. *(i.e.*, OnFill())
- Add the suffix "Handler" or optionally "ClassName + EventName" to your event handler methods. (*i.e.*, FillTankEventHandler() or AddWaterButtonClick())



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter13\WaterTank>watersystemapp
FillTankEventHandler: The water tank contains 1000 gallons!
FillTankEventHandler: The water tank contains 2000 gallons!
FillTankEventHandler: The water tank contains 3000 gallons!
FillTankEventHandler: The water tank contains 4000 gallons!
FillTankEventHandler: The water tank contains 5000 gallons!
FillTankEventHandler: The water tank contains 6000 gallons!
FillTankEventHandler: The water tank contains 7000 gallons!
FillTankEventHandler: The water tank contains 8000 gallons!
FullTankEventHandler: Draining the water tank!
DrainTankEventHandler: The water tank contains 8000 gallons!
DrainTankEventHandler: The water tank contains 7000 gallons!
DrainTankEventHandler: The water tank contains 6000 gallons!
DrainTankEventHandler: The water tank contains 5000 gallons!
DrainTankEventHandler: The water tank contains 4000 gallons!
DrainTankEventHandler: The water tank contains 3000 gallons!
DrainTankEventHandler: The water tank contains 2000 gallons!
EmptyTankEventHandler: Water tank has been drained! The water tank contains 1000 gallons!
```

When the pump starts draining the tank, the decreasing water level repeatedly triggers the Drain event. When the water level reaches the low-level set point, the Empty event fires and stops the pump.

Click the button to add water to the water tank. The rising water level repeatedly triggers the Fill event. When the water level reaches the high-level indicator set point, it fires the Full event.

Figure 13-6: Results of Running Example 13.11

Final Thoughts On Extending The EventArgs Class

In the previous two programming examples, I created custom event argument classes by extending the EventArgs class, but this was not strictly necessary, since I didn't use any of the functionality provided by the EventArgs class. In fact, the EventArgs class does nothing except provide a future evolutionary path for the .NET event API by serving as the base class for all the .NET event argument classes.

Summary

You need two logical components to implement the event processing model: 1) an event producer (*publisher*), and 2) an event consumer (*subscriber*). A *delegate* type specifies the authorized method signature for event handler methods. A delegate object maintains a list of event subscribers in the form of references to event handler methods. An event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. Event handler methods must conform to the method signature specified by an event's delegate type.

It helps to clarify the purpose of each component or method if you adopt the following or similar naming convention: add the suffix "EventArgs" to your event argument class name, add the suffix "EventHandler" to your event handler delegate names, add the prefix "On" to the event name for the method that fires the event, and finally, add the suffix "Handler" or optionally "ClassName + EventName" to your event handler methods.

It's not necessary to extend the System.EventArgs class to create custom event argument classes. The EventArgs class does nothing except provide a future evolutionary path for the .NET event API by serving as the base class for all the .NET event argument classes.

Skill-Building Exercises

1. **Compile and Execute Example Code:** Compile and execute the MinuteTick code given in Examples 13.1 through 13.5.

2. **Compile and Execute Example Code:** Compile and execute the automated water tank system code given in Examples 13.6 through 13.11.

3. **Create UML Sequence Diagram:** Create a UML sequence diagram for the Publisher.CountMinutes() method given in Example 13.3.

4. **Create UML Sequence Diagram:** Create a UML sequence diagram for the WaterTank.ChangeWaterLevel() method given in Example 13.10. You may actually need to create several diagrams to document the call to each WaterLevelSensor's WaterLevelChange() method. Pay particular attention to where the code starts to make recursive calls when the water tank is being drained.

5. **Have Some Fun:** Experiment with the automated water tank system code. Change the value of the Pump's pumping capacity and note the effects of the animation.

6. **API Drill:** Explore the .NET API and find all the delegates. List each delegate and describe its purpose.

Suggested Projects

1. **Oil Tanker Pumping System:** Revisit the oil tanker pumping system project given in Chapter 11, suggested project 5. Add a GUI that displays the tanks and their levels, the valves and their status (open or closed), and the pumps and their status. Utilize events to help monitor tank levels and to automatically start and stop the pumps. This will be a great project to let your imagination run wild!

2. **Refactor Code:** Using your knowledge of inheritance, redesign the WaterLevelSensor class given in Example 13.8 so that there are two separate classes called HighLevelSensor and LowLevelSensor. These two classes should derive from a common base class. Make the necessary modifications to the automated water tank system application.

3. **Modify Code:** Change the MinuteTick application presented in this chapter to respond to and display second ticks. (*i.e.*, every second vs. every minute).

Self-Test Questions

1. What two logical components are required to implement the event processing model?

2. What's the purpose of a delegate type?

3. Which keyword declares a new delegate type?

4. What's the relationship between delegates and events?

5. What's the relationship between delegates and event handler methods?

6. Which operator do you use register an event handler method with an object's event?

7. In what type of data structure does a delegate maintain its subscriber list?

8. In what order does the delegate make calls to its registered event handler methods?

9. What's the value of a delegate object that has no registered event handler methods?

10. Are you required to extend the EventArgs class to create custom event argument classes?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes