Chapter 5
Programming Structures

Though Java and C# share a common heritage and are similar in many respects, significant differences do exist between these two languages. For instance, C# introduced several new structures that are not supported by Java. These structures include indexers, delegates, attributes, and pointers. On the other hand, both languages support properties and events, albeit in different ways. In this chapter, the following programming structures will be introduced and code examples will be provided in the applicable language or languages:

- Properties
- Indexers
- Callbacks and Events
- Attributes
- Pointers

Properties

In object-oriented programming, a *property* is an intrinsic characteristic of an object. For instance, a `Car` class might contain properties of `make`, `model`, and `color`. These inherent characteristics specifically describe a particular car. The simplest method of implementing properties is to simply expose them as public fields as shown in Java here:

```
public class Car
{
  public String make;
  public String model;
  public String color;
}
```

These properties can then be accessed like this:

```
Car car = new Car();
car.make = "BMW";
car.model = "325i";
car.color = "white";

System.out.println("Car make: " + car.make);
System.out.println("Car model: " + car.model);
System.out.println("Car color: " + car.color);
```

Though properties can be exposed as public fields, there are several drawbacks to this approach. First, the value of a public field cannot be validated at the time it is set. For instance, suppose that each car is only available in the color of black or white. Using a public field, there is no way to enforce this restriction and the color may be set to an invalid value. Second, at times it

is desirable to make a property read-only or write-only. In other words, you may wish to allow a client to either read a property's value or set a property's value but not both. Public fields do not provide a means of making a property read-only or write-only. Last, public fields do not allow a property's value to be dynamically generated based on other properties or formatted in some manner each time it is read. For example, imagine that the `Car` class contains a property called `description` that combines the values of the `make`, `model`, and `color` properties. Using public fields, there is no way to dynamically generate a property's value at the time it is read.

Java and C# overcome the limitations of public field properties each in their own way. Java supports properties through adherence to a well-defined method naming convention. C#, on the other hand, formally supports properties within the language itself. Both approaches allow for property validation, read-only and write-only properties, and dynamically generated and formatted property values.

Java

Properties were added to Java as somewhat of an after-thought. The language itself does not formally support them. Regardless, Java properties are simple to implement, flexible, and very effective. A read/write property is implemented in Java as a corresponding pair of `get`*`Xxx`*`()` (a.k.a., *getter*) and `set`*`Xxx`*`()` (a.k.a., *setter*) methods where *Xxx* represents the name of the property (beginning with a capital letter). Each property's value is stored in a private variable that is accessible only through its corresponding `get`*`Xxx`*`()` and `set`*`Xxx`*`()` methods. These methods are commonly referred to as *accessor methods*. Using accessor methods, property values can be validated, calculated, or formatted dynamically when they are read or set. To illustrate, Listing 4.1 presents a `Car` class that implements the `make`, `model`, `color`, and `description` properties.

Listing 4.1 ***Car Class with Make, Model, Color, and Description Properties (Java)***

```
public class Car
{
  private String make;
  private String model;
  private String color;


  public void setMake(String make)
  {
    this.make = make.toUpperCase();
  }


  public String getMake()
  {
    return make;
  }


  public void setModel(String model)
  {
    this.model = model;
  }


  public String getModel()
  {
    return model;
```

```
  }


  public void setColor(String color)
  {
    if (color.equals("white") || color.equals("black"))
      this.color = color;
  }


  public String getColor()
  {
    return color;
  }


  public String getDescription()
  {
    return color + " " + make + " " + model;
  }
}
```

Listing 4.1 demonstrates several common property techniques. First, notice that the `make` property is converted to uppercase before it is stored. This type of property formatting is very common. Of course, it would also have been possible to store the property in lowercase and convert it to uppercase before it is returned by the `getMake()` method. The former approach was chosen since it is slightly more efficient. By converting the `make` value to uppercase when the property is set, the conversion takes place only once rather than every time the property is read. Second, the `color` property restricts valid colors to black and white. Any other setting will be ignored. In this way, it is not possible to set a car's color to an invalid value. Last, the `description` property demonstrates a read-only property whose value is dynamically generated at the time it is read. This property generates its value based on the values of the `color`, `make`, and `model` properties. Additionally, notice that there is no way to set the `description` property. This property was made read-only by not implementing a `setDescription()` method. Likewise, a property can be made write-only by implementing its setter method without a corresponding getter method. The `Car` class presented in Listing 4.1 can be used like this:

```
Car car = new Car();
car.setMake("BMW");
car.setModel("325i");
car.setColor("white");

System.out.println("Car make: " + car.getMake());
System.out.println("Car model: " + car.getModel());
System.out.println("Car color: " + car.getColor());
System.out.println("Car description: " + car.getDescription());
```

In addition to accessing properties directly from code, through a process called *introspection*, Java properties can be discovered and exposed by a development tool. For instance, let's say that you created a graphical component that displays formatted text. It is possible for a Java development tool to read the properties supported by your component, such as `font`, `size`, `color`, and `style`, and allow the user to set these properties from within the development environment without any coding. Java components that expose their properties in this manner are commonly referred to as *JavaBeans*.

C#

Properties are an integral part of the C# language. Unlike Java's method naming convention (i.e., `get` and `set` accessor methods), C# defines a formal syntax for properties that uses `get` and `set` keywords. To illustrate, Listing 4.2 presents the C# equivalent to the Java `Car` class shown in Listing 4.1.

Listing 4.2 ***Car Class with Make, Model, Color, and Description Properties (C#)***

```
public class Car
{
  private string make;
  private string model;
  private string color;


  public string Make
  {
    get
    {
      return make;
    }
    set
    {
      make = value.ToUpper();
    }
  }


  public string Model
  {
    get
    {
      return model;
    }
    set
    {
      model = value;
    }
  }


  public string Color
  {
    get
    {
      return color;
    }
    set
    {
      if (value == "white" || value == "black")
        color = value;
    }
  }


  public string Description
  {
    get
    {
      return color + " " + make + " " + model;
```

```
    }
  }
}
```

As you can see, C# properties differ significantly from their Java counterparts. First, C# properties are declared similar to public fields by specifying their accessibility (e.g., `private`, `protected`, `internal`, or `public`), data type (e.g., `string`, `int`, `long`, etc.), and name. However, unlike public fields, C# properties are followed by a special code block containing `get` and/or `set` accessor blocks. The `get` and `set` blocks are automatically invoked when the property is read or set, respectively. Notice that a special parameter named `value` is made available to the code within the `set` accessor block. Being of the same data type as the property, the `value` parameter contains the value to which the property should be set. Properties can be made read-only or write-only by including only the appropriate accessor block. For instance, to make a property read-only, only the `get` accessor block should be included (excluding the `set` block). In this way, the property's value can be read but cannot be set.

In addition to looking very different from Java properties, C# properties are accessed differently. While Java properties are accessed like methods, C# properties are accessed as if they were public fields. For example, C# properties are used like this:

```
Car car = new Car();
car.Make = "BMW";
car.Model = "325i";
car.Color = "white";

Console.WriteLine("Car make: {0}", car.Make);
Console.WriteLine("Car model: {0}", car.Model);
Console.WriteLine("Car color: {0}", car.Color);
Console.WriteLine("Car description: {0}", car.Description);
```

Notice how the C# properties are accessed exactly like a public field. At times, this feature can greatly simplify your source code. For instance, consider how you would increment the value of an integer property named `count` in Java:

```
car.setCount(car.getCount() + 1);
```

The same functionality in C# would look like this:

```
car.Count++;
```

As you can see, working with C# properties can be a lot more convenient. There is one caveat though. Since it is not completely obvious that custom code is being executed when a C# property is accessed, it can be more difficult to locate performance problems in your code. For example, there may be a call to a database in the following code. Can you find it?

```
Console.WriteLine("Name: {0}", customer.Name);
Console.WriteLine("Total Orders: {0}", customer.TotalOrders);
```

The database call is executed in the `TotalOrders` property's `get` accessor block. Since the `TotalOrders` property wasn't accessed like a method call (e.g., `customer.getTotalOrders()`), it is not readily apparent that significant additional processing may be taking place. For this reason, you should typically avoid performing expensive computations within a C# property's accessor blocks.

NOTE

With Java properties, it is possible to declare different accessibility levels to a property's `get` and `set` accessor methods. For instance, you could declare the `set` method as `protected` and the `get` method as `public`. This is not possible in C#. Since accessibility is declared at the property level, all C# properties grant the same accessibility to both the `get` and `set` accessor blocks.

Indexers (C#)

When working with a class that represents a collection of objects, it may be convenient to access the class as if it were an array. C# supports this behavior through a feature known as an *indexer*. Indexers allow classes to define the manner in which they expose elements of an internal collection. For example, assume that you have a class called `Planets` that contains a collection (or array) of `Planet` objects that look like this:

```
public class Planet
{
  private string name; //name of planet
  private int diameter; //diameter in kilometers


  public Planet(string name, int diameter)
  {
    this.name = name;
    this.diameter = diameter;
  }


  public string Name
  {
    get
    {
      return name;
    }

    set
    {
      name = value;
    }
  }


  public int Diameter
  {
    get
    {
      return diameter;
    }

    set
    {
      diameter = value;
    }
  }
}
```

Traditionally, the `Planets` collection would expose methods to access each planet according to its index like this:

```
public class Planets
{
  private Planet[] planets;


  public Planets(int numPlanets)
  {
    planets = new Planet[numPlanets];
  }


  public Planet Get(int index)
  {
    if (index < planets.Length)
      return planets[index];
    else
      return null;
  }


  public void Set(int index, Planet planet)
  {
    if (index < planets.Length)
      planets[index] = planet;
  }
}
```

Since it doesn't support indexers, this is typically how you would access elements of a collection in Java. Using this coding style, elements of the `planets` array can be accessed as follows:

```
public static void Main()
{
  Planets planets = new Planets(3);

  planets.Set(0, new Planet("Mercury", 4880));
  planets.Set(1, new Planet("Venus", 12103));
  planets.Set(2, new Planet("Earth", 12756));

  Console.WriteLine(planets.Get(0).Name);
  Console.WriteLine(planets.Get(1).Name);
  Console.WriteLine(planets.Get(2).Name);
}
```

Now that we have seen the traditional method, let's take a look out how this same functionality can be accomplished using the C# indexer construct. List 4.3 demonstrates a `Planets` class that replaces the traditional `Get()` and `Set()` methods with an indexer:

Listing 4.3 ***Planets class that implements an indexer (C#)***

```
public class Planets
{
  private Planet[] planets;


  public Planets(int numPlanets)
  {
```

```
    planets = new Planet[numPlanets];
  }


  /// <summary>
  /// Define an indexer for the Planets class.
  /// </summary>
  public Planet this[int index]
  {
    get
    {
      if (index < planets.Length)
        return planets[index];
      else
        return null;
    }

    set
    {
      if (index < planets.Length)
        planets[index] = value;
    }
  }
}
```

As you can see from Listing 4.3, an indexer is very similar to a property. The primary difference is that a property is assigned a specific name while an indexer uses the `this` keyword. The `this` keyword indicates that the indexer is accessed directly from the object without specifying the name of a property. Thus, syntactically speaking, elements of an indexer class can be accessed as if the class was an array. For example, the following code accesses elements of the `Planets` class using an indexer.

```
public static void Main()
{
  Planets planets = new Planets(3);

  planets[0] = new Planet("Mercury", 4880);
  planets[1] = new Planet("Venus", 12103);
  planets[2] = new Planet("Earth", 12756);

  Console.WriteLine(planets[0].Name);
  Console.WriteLine(planets[1].Name);
  Console.WriteLine(planets[2].Name);
}
```

Classes are not limited to a single indexer. Rather, any number of indexers is supported as long as each indexer has a unique signature. That is, each indexer must accept a different type or number of parameters. For instance, Listing 4.4 demonstrates a `Planets` class that implements two indexers—one that accepts an integer and one that accepts a string.

Listing 4.3 ***Planets class that implements two indexers (C#)***

```
public class Planets
{
  private Planet[] planets;


  public Planets(int numPlanets)
  {
```

```
    planets = new Planet[numPlanets];
  }


  /// <summary>
  /// Integer indexer.
  /// </summary>
  public Planet this[int index]
  {
    get
    {
      if (index < planets.Length)
        return planets[index];
      else
        return null;
    }

    set
    {
      if (index < planets.Length)
        planets[index] = value;
    }
  }


  /// <summary>
  /// String indexer.
  /// </summary>
  public Planet this[String name]
  {
    get
    {
      foreach (Planet p in planets)
      {
        if (p.Name == name)
          return p;
      }

      return null;
    }
  }
}
```

The two indexers implemented in Listing 4.4 can be accessed as follows:

```
public static void Main()
{
  Planets planets = new Planets(3);

  planets[0] = new Planet("Mercury", 4880);
  planets[1] = new Planet("Venus", 12103);
  planets[2] = new Planet("Earth", 12756);

  //retrieve elements using integer index
  Console.WriteLine("{0}, {1}", planets[0].Name,
    planets[0].Diameter);
  Console.WriteLine("{0}, {1}", planets[1].Name,
    planets[1].Diameter);
  Console.WriteLine("{0}, {1}", planets[2].Name,
    planets[2].Diameter);
```

```
  //retrieve elements using string index
  Console.WriteLine("{0}, {1}", planets["Mercury"].Name,
    planets["Mercury"].Diameter);
  Console.WriteLine("{0}, {1}", planets["Venus"].Name,
    planets["Venus"].Diameter);
  Console.WriteLine("{0}, {1}", planets["Earth"].Name,
    planets["Earth"].Diameter);
}
```

Furthermore, indexers are not limited to a single parameter. A multiple parameter indexer can be defined as follows:

```
public Planet this[int x, int y]
{
  get
  {
    //get implementation...
  }

  set
  {
    //set implementation...
  }
}
```

Indexers offer an elegant and consistent method of accessing elements encapsulated within a class. Whenever a class represents a collection of objects, consider implementing an indexer to provide easy access to its internal elements.

Callbacks and Events

A *callback* is a function or method that a program indicates should be called when a certain event takes place (such as a user pressing a key on the keyboard or clicking a mouse button). Callbacks enable event-driven programming by notifying interested programs when a particular event occurs rather than requiring each program to poll for the event itself. Callbacks are implemented using the publish/subscribe model. That is, programs that wish to be notified when a certain event occurs must "subscribe" to the event by registering a particular method with the event source (or "publisher"). In turn, when the event takes place, the event source invokes each registered method. In this manner, programs can be efficiently notified without having to constantly poll for the event of interest.

Java and C# implement callbacks and handle events quite differently. For instance, Java uses listener interfaces to create callbacks and handle events while C# employs specialized `delegate` and `event` types for this purpose. Though the merits of each approach are often debated, by and large, they are equally effective.

Java

Java utilizes listener interfaces to handle callbacks and events. As discussed in Chapter 4, an interface is an object-oriented programming construct that defines a collection of methods exposed by a class. A *listener interface* is simply an interface that defines the method that should be called whenever a particular event occurs. All classes that wish to be notified when a certain

event occurs must implement the appropriate listener interface and register themselves with the object that generates the event (i.e., the event source). Java convention dictates that each event source should expose a method or methods of the form `add`*`Xxx`*`Listener()` (where *`Xxx`* represents the name of the event). If a class wishes to be notified when an event occurs, it simply implements the appropriate listener interface and then passes a reference to itself to the event source via the `add`*`Xxx`*`Listener()` method. The event source maintains a collection of references to all registered listeners and, when an event occurs, notifies each listener by calling the method defined by the listener interface. To summarize, the entire process works like this:

1. The event source defines a listener interface that must be implemented by all classes that wish to be notified when a particular event occurs.
2. A class that wishes to be notified implements the listener interface that corresponds to the event of interest. We can refer to the implemented method as the *listener method* and an instance of the listener class as the *listener object*.
3. A listener object registers itself with the event source by calling the source's `add`*`Xxx`*`Listener()` method that corresponds to the event of interest.
4. The event source maintains a collection of all objects that wish to be notified when a particular event occurs.
5. When the event of interest occurs, the event source iterates through its collection of listener objects, calling the appropriate listener method on each object.

Now that we have discussed the Java approach to callbacks and events, let's take a look at some code. Listing 4.4 presents a typical event source.

Listing 4.4 ***Basic event source (Java)***

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

/**
 * Notifies listeners after random time interval.
 */
public class RandomTimer
{
  private ArrayList listeners = new ArrayList();


  /**
   * Adds object to listener collection.
   */
  public void addTimerListener(RandomTimerListener listener)
  {
    listeners.add(listener);
  }


  /**
   * Removes object from listener collection.
   */
  public void removeTimerListener(RandomTimerListener listener)
  {
    listeners.remove(listener);
  }
```

```
  /**
   * Starts the event timer.
   */
  public void startTimer()
  {
    Random rnd = new Random(System.currentTimeMillis());
    int seconds = rnd.nextInt(10);

    try
    {
      //sleep for 0 to 10 seconds and then notify listeners
      Thread.sleep(seconds*1000);

      notifyListeners(seconds);
    }
    catch (InterruptedException ignored) {}
  }


  /**
   * Iterates through listeners collection, notifying each one.
   */
  private void notifyListeners(int seconds)
  {
    for (Iterator it = listeners.iterator(); it.hasNext();)
    {
      RandomTimerListener rtl = (RandomTimerListener)it.next();
      rtl.randomTimerEvent(seconds);
    }
  }


  /**
   * Listener interface.
   */
  public static interface RandomTimerListener
  {
    public void randomTimerEvent(int seconds);
  }
}
```

Listing 4.5 demonstrates a simple program that registers itself with the event source and writes a message to standard out when it is notified that the event has occurred.

Listing 4.5 ***Simple listener class (Java)***

```
/**
 * Listener class implements the RandomTimerListener interface.
 */
public class TimerListener implements
  RandomTimer.RandomTimerListener
{
  /**
   * Instantiates the RandomTimer event source and an instance
   * of itself, registers as a listener with the event source,
   * and starts the event source timer.
   */
  public static void main(String args[])
  {
    RandomTimer randomTimer = new RandomTimer();
    TimerListener timerTester = new TimerListener();
```

```
    randomTimer.addTimerListener(timerTester);

    System.out.println("Starting timer...");
    randomTimer.startTimer();
  }


  /**
   * Listener method called by event source. The signature of
   * this method is defined by the listener interface.
   */
  public void randomTimerEvent(int seconds)
  {
    System.out.println("Timer fired after "+ seconds +
      " seconds.");
  }
}
```

Listings 4.4 and 4.5 demonstrate how easy it is to create an event source and event listener that adheres to the standard Java event model. Now that we know how to build a custom event source, let's examine how to use one of Java's most common event sources—the `JButton`. In contrast to our previous example, when listening for events generated by any of the standard Java event sources (such as a user interface element), an inner class is typically employed. Listing 4.6 illustrates how an inner class can be used as an event listener.

Listing 4.6 ***Event listener using an inner class (Java)***

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Creates a window that contains a button. When the button is
 * clicked, an inner class is notified and the program exits.
 */
public class ButtonTester extends JFrame
{
  JButton button;


  /**
   * Create button, register inner class with button so that it
   * is called when the button click event takes place, and add
   * button to the JFrame window.
   */
  public ButtonTester()
  {
    button = new JButton("Exit");
    button.addActionListener(new ClickEvent());
    this.getContentPane().add(button);
  }


  /**
   * Instantiate this class and make the frame visible.
   */
  public static void main(String[] args)
  {
    ButtonTester bl = new ButtonTester();
```

```
    bl.pack(); //resizes frame to fit button
    bl.setVisible(true);
  }


  /**
   * Inner class that implements the listener interface. An
   * instance of this class is called when the button click
   * event takes place.
   */
  private class ClickEvent implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      System.exit(0); //terminate program when button is clicked
    }
  }
}
```

Using inner classes, a class can listen for events without having to implement any listener interfaces (they are implemented by the inner classes). This approach often results in a more manageable and elegant solution, especially when the class is interested in a large number of events.

As you can see, the Java designers implemented a comprehensive callback and event model using only basic object-oriented constructs—classes and interfaces. This decision is in stark contrast to the C# approach that introduces two specialized constructs to support callbacks and events—the `delegate` and `event` types.

C#

C# defines two types specifically designed to handle callbacks and events. The `delegate` type facilitates callbacks and event handling while the `event` type provides a formal method of declaring events.

Delegates

Similar to a C/C++ function pointer, a *`delegate`* in C# provides a reference, or an alias, to a particular method. In other words, a method reference can be encapsulated within a `delegate` and passed as a parameter to another class from where it may be called. For this reason, delegates are ideally suited for implementing callbacks.

In many respects, a `delegate` is similar to both a class and an object. For example, a delegate template defines the signature of the methods to which its instances can point (analogous to a class) while an instance of the `delegate` provides a reference to a specific method or methods (analogous to an object). Though both are commonly referred to simply as delegates, when necessary, we will distinguish between the two using the terms *delegate type* for the delegate template and *delegate instance* for a particular instance of the `delegate`. To illustrate, the following code declares a delegate type capable of referencing methods that accept two integer parameters and return an integer:

```
public delegate int AddMethod(int x, int y);
```

Once the delegate type is defined, a delegate instance that references an existing method can be created. For instance, assuming that a static method called `AddValues()` already exists within the class and its signature matches that of the delegate type, a delegate instance that points to this method can be created as follows:

```
AddMethod add = new AddMethod(AddValues);
```

Since the delegate already knows the method's signature, only the name of the method is passed to the delegate's constructor. If the signatures do not match, a compile-time exception is thrown. Once the delegate instance is created, the `AddValues()` method can be called through the delegate like this:

```
int result1 = add(1, 2); //uses delegate to call original method

//the delegate call is equivalent to this direct method call
int result2 = AddValues(1, 2);
```

If the referenced method is not declared `static`, the object instance to which it belongs must be specified like this:

```
AddMethod add = new AddMethod(this.AddValues);
```

A delegate type is similar to a class or interface in that it defines a new type of object and can be declared either inside or outside of a class. Typically, a delegate type that may be used by multiple classes is declared as `public` outside of any class or within an appropriate container class. On the other hand, a delegate type that is only used within a single class may be declared `private` within the class. The following code demonstrates how a delegate type can be declared as `public` or `private` either inside or outside of a class.

```
public delegate int PublicDelegateOutsideClass(int x, int y);

public class Test
{
  public delegate int PublicDelegateInsideClass(int x, int y);
  private delegate int privateDelegateType(int x, int y);
}
```

A public delegate type declared within a class is automatically static. That is, it is referenced using the name of the class like this:

```
Test.PublicDelegate2 add = new Test.PublicDelegate2(AddValues);
```

Unlike a delegate type, a delegate instance can only be declared within a class or `struct` like this:

```
public delegate int DelegateType(int x, int y);

public class Test
{
  public DelegateType DelegateInstance = new DelegateType(Add);

  public static int Add(int x, int y)
  {
    return x + y;
  }
}
```

Now let's take a look at delegates in action. Listings 4.7 and 4.8 are equivalent to the Java versions shown in Listings 4.4 and 4.5, respectively, and demonstrate how delegates can be used as a callback mechanism.

Listing 4.7 ***Basic event source (C#)***

```
using System;
using System.Collections;

/// <summary>
/// Notifies delegates after random time interval.
/// </summary>
public class RandomTimer
{
  public delegate void TimerCallback(int seconds);
  private ArrayList timerCallbacks = new ArrayList();


  /// <summary>
  /// Adds a delegate to the callback collection.
  /// </summary>
  public void AddCallback(TimerCallback callback)
  {
    timerCallbacks.Add(callback);
  }


  /// <summary>
  /// Removes a delegate from the callback collection.
  /// </summary>
  public void RemoveCallback(TimerCallback callback)
  {
    timerCallbacks.Remove(callback);
  }


  /// <summary>
  /// Starts the random timer and calls delegates.
  /// </summary>
  public void StartTimer()
  {
    Random rnd = new Random((int)DateTime.Now.Ticks);
    int seconds = rnd.Next(11);

    //sleep for 0 to 10 seconds and then notify listeners
    System.Threading.Thread.Sleep(seconds*1000);

    notifyCallbacks(seconds);
  }


  /// <summary>
  /// Iterates through callbacks collection, invoking each one.
  /// </summary>
  private void notifyCallbacks(int seconds)
  {
    foreach (TimerCallback callback in timerCallbacks)
    {
      callback(seconds);
    }
  }
}
```

Listing 4.8 ***Simple event observer (C#)***

```
using System;

/// <summary>
/// Uses a delegate to be notified when the RandomTimer object's
/// timer event fires.
/// </summary>
public class TimerTest
{
  /// <summary>
  /// Instantiates a RandomTimer object, creates a delegate
  /// instance pointing to its local OnRandomTimer() method, and
  /// calls the RandomTimer object's StartTimer() method
  /// (passing the delegate for callback purposes).
  /// </summary>
  public static void Main()
  {
    RandomTimer randomTimer = new RandomTimer();

    RandomTimer.TimerCallback callback =
      new RandomTimer.TimerCallback(OnRandomTimer);

    randomTimer.AddCallback(callback);

    Console.WriteLine("Starting timer...");
    randomTimer.StartTimer();
  }


  /// <summary>
  /// Event handler method called by event source via the
  /// delegate.
  /// </summary>
  public static void OnRandomTimer(int seconds)
  {
    Console.WriteLine("Timer fired after {0} seconds.",
      seconds);
  }
}
```

In order to closely match the Java version, the `RandomTimer` class in C# maintains a collection of delegates so that multiple methods may be called when a single event occurs. However, in C#, there is a simpler way to implement this feature. The simpler method is based on the fact that multiple method references can be added to a single delegate instance. A delegate instance that contains multiple method references is known as a *multicast delegate*. A multicast delegate can be created using the += operator like this:

```
public delegate void NewHireNotify(string name);

NewHireNotify newHire = new NewHireNotify(UpdateHR);
newHire += new NewHireNotify(UpdatePayRoll);
newHire += new NewHireNotify(UpdateIT);

newHire("Kennedy Callaway"); //calls three methods
```

The += operator is convenient but we could have also created the same result using just the + operator as follows:

```
public delegate void NewHireNotify(string name);

NewHireNotify newHireHR = new NewHireNotify(UpdateHR);
NewHireNotify newHirePR = new NewHireNotify(UpdatePayRoll);
NewHireNotify newHireIT = new NewHireNotify(UpdateIT);

NewHireNotify newHire = newHireHR + newHirePR + newHireIT;

newHire("Kennedy Callaway"); //calls three methods
```

Thus, the `RandomTimer` class can be simplified by replacing the delegates collection with a single multicast delegate as shown in Listing 4.9.

Listing 4.9 ***RandomTimer class using a multicast delegate (C#)***

```
using System;

/// <summary>
/// Notifies delegate after random time interval.
/// </summary>
public class RandomTimer
{
  public delegate void TimerCallback(int seconds);
  public TimerCallback TimerCallbacks; //multicast delegate


  /// <summary>
  /// Starts the random timer and calls multicast delegate.
  /// </summary>
  public void StartTimer()
  {
    Random rnd = new Random((int)DateTime.Now.Ticks);
    int seconds = rnd.Next(11);

    //sleep for 0 to 10 seconds and then notify listeners
    System.Threading.Thread.Sleep(seconds*1000);

    TimerCallbacks(seconds);
  }
}
```

Notice that Listing 4.9 no longer contains the `AddCallback()` and `RemoveCallback()` methods. Since the `TimerCallbacks` multicast delegate is declared `public`, these methods are no longer necessary. Instead, new callbacks can be registered by simply adding delegates to the multicast delegate as demonstrated in Listing 4.10. Delegates are called in the order that they were added to the multicast delegate. As you can see, multicast delegates provide an elegant solution that can significantly simplify the event source and event observer classes.

Listing 4.10 ***TimerTest class using a multicast delegate (C#)***

```
using System;

/// <summary>
/// Uses a delegate to be notified when the RandomTimer object's
/// timer event fires.
/// </summary>
public class TimerTest
{
  /// <summary>
  /// Instantiates a RandomTimer object, creates a delegate
```

```
  /// instance pointing to its local OnRandomTimer() method, and
  /// calls the RandomTimer object's StartTimer() method
  /// (passing the delegate for callback purposes).
  /// </summary>
  public static void Main()
  {
    RandomTimer randomTimer = new RandomTimer();

    RandomTimer.TimerCallback callback =
      new RandomTimer.TimerCallback(OnRandomTimer);

    //add new delegate to multicast delegate
    randomTimer.TimerCallbacks += callback;

    Console.WriteLine("Starting timer...");
    randomTimer.StartTimer();
  }


  /// <summary>
  /// Event handler method called by event source via the
  /// delegate.
  /// </summary>
  public static void OnRandomTimer(int seconds)
  {
    Console.WriteLine("Timer fired after {0} seconds.",
      seconds);
    Console.ReadLine();
  }
}
```

There are a couple of caveats to consider when implementing a multicast delegate. First, if a multicast delegate's signature contains a return value, only the value of the last method will be returned. For this reason, multicast delegates are usually defined using a `void` return type. Second, if an uncaught exception is thrown by one of the methods, the error is immediately propagated up the call stack and the remaining methods are not called.

Events

Since event handling is integral to .NET programming, the C# designers chose to include a formal `event` type as part of the language. The `event` type functions nearly identically to the public delegate instance, `TimerCallbacks`, demonstrated in Listings 4.9 and 4.10. There are, however, two notable differences between them. First, a public delegate instance exposes all of the delegate's public methods to the client. This may not always be desirable. In contrast, the `event` type implicitly creates a private multicast delegate, exposing only its `+=` and `-=` operators. Second, the `event` type provides a formal declaration conveying that the object does, in fact, represent an event. A delegate may be used for purposes other than pure event handling. Formally declared events can be easily located through reflection and presented to the user within an integrated development environment.

Within an event source, an `event` is declared just like a delegate instance except that it includes the `event` keyword in its declaration. For example, the following code defines a delegate type and then declares an `event` that accepts delegate instances of that type:

```
public delegate void TimerExpiredEventHandler(int seconds);
public event TimerExpiredEventHandler TimerExpired;
```

In other words, the `event` declaration implicitly creates a private multicast delegate instance to which an observer can add delegates of the type `TimerExpiredEventHandler` like this:

```
TimerExpired += new TimerExpiredEventHandler(TimesUp);

public static void TimesUp(int seconds)
{
  Console.WriteLine("{0} seconds has expired.", seconds);
}
```

Similar to a multicast delegate, the event source notifies all observers ("listeners" in Java) by invoking the `event` object as follows:

```
Random rnd = new Random((int)DateTime.Now.Ticks);
int seconds = rnd.Next(11);
System.Threading.Thread.Sleep(seconds*1000);

TimerExpired(seconds); //invoke event to notify observers
```

Now let's put it all together. Listing 4.11 demonstrates an event source that notifies observers using the `event` type.

Listing 4.11 ***Event source class using the event type (C#)***

```
using System;

/// <summary>
/// Invokes an event after random time interval.
/// </summary>
public class RandomTimer
{
  public delegate void TimerExpiredEventHandler(int seconds);
  public event TimerExpiredEventHandler TimerExpired;


  /// <summary>
  /// Starts the random timer and invokes the event.
  /// </summary>
  public void StartTimer()
  {
    Random rnd = new Random((int)DateTime.Now.Ticks);
    int seconds = rnd.Next(11);

    //sleep for 0 to 10 seconds and then notify listeners
    System.Threading.Thread.Sleep(seconds*1000);

    TimerExpired(seconds);
  }
}
```

Listing 4.12 demonstrates how an observer tells the event source that it wishes to be notified when a specified event occurs. Delegates assigned to an event are called in the order in which they were added.

> **NOTE**
> Events and multicast delegates are single threaded. When invoked, the first assigned delegate is called and the event or multicast delegate blocks until control is returned. At this point, the second assigned

delegate is called and so on. For this reason, event observers should spawn a thread to execute any lengthy processing. This will ensure that all observers are notified in a timely manner.

Listing 4.12 ***Event observer class (C#)***

```
/// <summary>
/// Uses an event to be notified when the RandomTimer object's
/// TimerExpired event fires.
/// </summary>
public class TimerTest
{
  /// <summary>
  /// Instantiates a RandomTimer object, creates a delegate
  /// instance pointing to its local OnTimerExpired() method,
  /// and calls the RandomTimer object's StartTimer() method.
  /// </summary>
  public static void Main()
  {
    RandomTimer randomTimer = new RandomTimer();

    randomTimer.TimerExpired +=
      new RandomTimer.TimerExpiredEventHandler(TimesUp);

    Console.WriteLine("Starting timer...");
    randomTimer.StartTimer();
  }


  /// <summary>
  /// Event handler method called by event source.
  /// </summary>
  public static void TimesUp(int seconds)
  {
    Console.WriteLine("Timer fired after {0} seconds.",
      seconds);
    Console.ReadLine();
  }
}
```

For the sake of simplicity, the previous code examples have not strictly adhered to the .NET standard for handling events. Previous event handler methods, such as the `TimesUp()` method in Listing 4.12, have used any name and signature deemed appropriate for their purpose. However, according to .NET convention, all event handler methods (i.e., methods that are called by the event source) should adhere to the following naming convention and signature:

```
void ObjectName_EventName(object sender, System.EventArgs e)
```

To illustrate, assume that an observer class instantiates a reference to an event source like this:

```
RandomTimer randomTimer = new RandomTimer();
```

If the `randomTimer` object exposes an event called `TimerExpired`, the event handler method in the observer class should be named as follows:

```
public void randomTimer_TimerExpired(object sender, EventArgs e)
```

First, let's take a look at the method name. The `randomTimer_TimerExpired` name indicates that this method is an event handler for the `randomTimer` object's `TimerExpired` event. This naming convention makes event handling code much easier to understand and maintain.

Next, let's examine the method signature. The standard event handler signature consists of two parameters, `object` and `System.EventArgs` (or a subclass of `System.EventArgs`), and a `void` return type. The `object` parameter represents the event source. Since a single event handler can be invoked by multiple event sources, this parameter is necessary to allow the handler to determine the event's origin. For instance, an event observer could assign a single event handler to multiple event sources as shown in Listing 4.13. This listing also demonstrates how the event's origin can be determined using the event handler's `object` parameter.

Listing 4.13 ***Event observer class (C#)***

```
using System;

/// <summary>
/// Creates two RandomTimer instances and starts both timers.
/// This class assumes that the StartTimer() methods for each
/// RandomTimer are running on separate threads.
/// </summary>
public class TimerTest
{
  private static RandomTimer randomTimer1;
  private static RandomTimer randomTimer2;


  /// <summary>
  /// Instantiates two RandomTimer object, assigns the same
  /// event handler to both objects, and starts each object's
  /// timer.
  /// </summary>
  public static void Main()
  {
    randomTimer1 = new RandomTimer();
    randomTimer2 = new RandomTimer();

    //get shared event handler
    RandomTimer.TimerExpiredEventHandler timerHandler =
      new RandomTimer.TimerExpiredEventHandler(TimesUp);

    //register event handler with both timers
    randomTimer1.TimerExpired += timerHandler;
    randomTimer2.TimerExpired += timerHandler;

    Console.WriteLine("Starting timers...");
    randomTimer1.StartTimer();
    randomTimer2.StartTimer();
  }


  /// <summary>
  /// Event handler method called by event source. Determines
  /// event's origin using the sender object.
  /// </summary>
  public static void TimesUp(object sender, EventArgs e)
  {
    if (sender == randomTimer1)
      Console.WriteLine("Timer1 timer expired.");
```

```
    else if (sender == randomTimer2)
      Console.WriteLine("Timer2 timer expired.");
  }
}
```

Following the `object` parameter, the `EventArgs` parameter is used to convey specific information about the event that has just occurred. Since the `System.EventArgs` base class contains no event-specific information, it is only used by events that do not communicate state information. To convey state information to the event handler, the `System.EventArgs` class must be extended to include event-specific information. To illustrate this point, examine the differences between the `TimesUp()` event handlers in Listings 4.12 and 4.13. Notice that the handler in Listing 4.12 is passed an integer indicating the timer delay but Listing 4.13 does not receive this information. To pass this event-specific information while adhering to the standard .NET event handler signature, the `System.EventArgs` parameter must be extended as shown in Listing 4.14. Notice how the event-specific `TimerExpiredEventArgs` class includes the `Seconds` property.

Listing 4.14 ***Custom EventArgs class includes timer information (C#)***

```
using System;

/// <summary>
/// Adds the Seconds property to the standard EventArgs class.
/// </summary>
public class TimerExpiredEventArgs: EventArgs
{
  private int seconds;


  public TimerExpiredEventArgs(int seconds): base()
  {
    this.seconds = seconds;
  }


  public int Seconds
  {
    get
    {
      return seconds;
    }
  }
}
```

The `RandomTimer` event source and `TimerTest` event observer classes can then be modified to use the new `TimerExpiredEventArgs` class as shown in Listing 4.15.

Listing 4.15 ***Event source and observer that uses a custom EventArgs class (C#)***

```
using System;

/// <summary>
/// Invokes event after random time interval.
/// </summary>
public class RandomTimer
{
  public delegate void TimerExpiredEventHandler(object sender,
    TimerExpiredEventArgs e);
  public event TimerExpiredEventHandler TimerExpired;
```

```
  /// <summary>
  /// Starts the random timer and invokes event.
  /// </summary>
  public void StartTimer()
  {
    Random rnd = new Random((int)DateTime.Now.Ticks);
    int seconds = rnd.Next(11);

    //sleep for 0 to 10 seconds and then notify listeners
    System.Threading.Thread.Sleep(seconds*1000);

    OnTimerExpired(new TimerExpiredEventArgs(seconds));
  }


  /// <summary>
  /// Allows base classes to fire event.
  /// </summary>
  protected void OnTimerExpired(TimerExpiredEventArgs e)
  {
    if (e != null)
      TimerExpired(this, e);
  }
}


/// <summary>
/// Uses a delegate to be notified when the RandomTimer object's
/// timer event fires.
/// </summary>
public class TimerTest
{
  /// <summary>
  /// Instantiates a RandomTimer object, creates a delegate
  /// instance pointing to its local randomTimer_TimerExpired()
  /// method, and calls the RandomTimer object's StartTimer()
  /// method.
  /// </summary>
  public static void Main()
  {
    RandomTimer randomTimer = new RandomTimer();

    randomTimer.TimerExpired +=
      new RandomTimer.TimerExpiredEventHandler(
      randomTimer_TimerExpired);

    Console.WriteLine("Starting timer...");
    randomTimer.StartTimer();
  }


  /// <summary>
  /// Event handler method called by event source.
  /// </summary>
  public static void randomTimer_TimerExpired(object sender,
    TimerExpiredEventArgs e)
  {
    Console.WriteLine("Timer fired after {0} seconds.",
      e.Seconds);
```

```
   }
}
```

Notice how the `randomTimer_TimerExpired()` event handler at the bottom of Listing 4.15 extracts the number of seconds expired from the event-specific `EventArgs` parameter.

Before concluding our discussion on events, there is one more .NET convention that should be mentioned. Notice the `OnTimerExpired()` method implemented in the `RandomTimer` class shown in Listing 4.15. By convention, .NET event sources should implement a `protected` method named `On`*`Xxx`*`()` for each event where *`Xxx`* is the name of the event. This method allows the event to be raised by classes that extend the event source.

As we have seen, callbacks and events are implemented differently in Java and C#. Perhaps the most salient difference between the two approaches is the fact that C# defines a formal `event` type while Java supports events through traditional classes and interfaces. As to which approach is better, it is simply a question of personal preference. In practice, both approaches are simple and effective.

Attributes (C#)

When a C# class is compiled, the compiler generates information describing each of its members (e.g., constructors, properties, methods, events, interfaces, etc.) and stores this information within the compiled file. This information, referred to as *metadata*, can then be examined programmatically through a process called reflection. Included in the compilation as part of the class's metadata, an *attribute* is an annotation that is attached to a particular entity (e.g., class, field, property, method). These annotations provide additional information and instructions to compilers, development environments, or any other programs that evaluate metadata. For example, attributes might convey to the compiler that a portion of code is obsolete or should be conditionally compiled. Similarly, an attribute could tell a development environment how a property should be displayed or convey to an application server the transaction support required by a particular method.

> **NOTE**
> Though not currently available, through Java Specification Request 175, work is under way to add the ability to specify custom metadata to the Java language. The functionality proposed by JSR 175 is similar to the provided by C# attributes.

Using Attributes

C# includes a large number of pre-defined attributes that can be attached to elements within your code. Attributes are designated within brackets (e.g., `[`*`AttributeName`*`]`) and immediately precede the entity to which they apply. For instance, the following `[Serializable]` attribute indicates that the associated class can be converted to a form that can be persisted to disk or transferred across the network:

```
[Serializable]
class MyClass
{
  //class implementation here...
}
```

An attribute is a special type of class that extends from `System.Attribute`. By convention, attribute classes end with the word `Attribute`. For instance, the `[Serializable]` attribute is defined in a class called `SerializableAttribute`. When the compiler encounters the `[Serializable]` attribute in source code, it searches for a class called `Serializable` that extends from `System.Attribute`. If not found, it searches for a class called `SerializableAttribute` that extends from `System.Attribute`. Since attributes are actually classes, you must either fully specify the namespace in which they reside like this:

```
[System.Serializable]
```

or import the namespace with the `using` statement like this:

```
using System;
[Serializable]
```

The `using` statement must come before all code elements, including attributes. Only comments are allowed to precede a `using` statement.

Depending on the attribute, parameters may or may not be required as part of the attribute declaration. Similar to methods, attribute parameters are specified using a comma-delimited list of values enclosed in parentheses. For example, the `[Obsolete]` attribute (defined in the `System` namespace) indicates that an entity is obsolete and its use is discouraged. The C# compiler will generate a warning when an obsolete entity is referenced. Though the `[Obsolete]` attribute can be used without parameters to generate a generic message, a custom warning message can be specified using a parameter as demonstrated in Listing 4.16.

Listing 4.16 ***Class includes an obsolete method (C#)***

```
using System;

public class MathStuff
{
  [Obsolete("Use AddNumbers() instead.")]
  public int AddTwoNumbers(int x, int y)
  {
    return x + y;
  }


  public int AddNumbers(params int[] nums)
  {
    int total = 0;

    foreach (int num in nums)
    {
      total += num;
    }

    return total;
  }
}
```

Listing 4.16 indicates that if the `AddTwoNumbers()` method is accessed, the compiler should generate a warning that includes the message "`Use AddNumbers() instead.`" Additionally, by adding a second parameter to the `[Obsolete()]` attribute, the compiler can be directed to generate an error (rather than a warning) when an obsolete method is used. To ensure that an

obsolete entity cannot be accessed, pass "`true`" as the second parameter of the `[Obsolete()]` attribute like this:

```
[Obsolete("Use AddNumbers() instead.", true)]
```

Finally, attribute properties can be set from within the parameter list by specifying the property name and value. For instance, the following attribute passes a normal parameter plus a property setting:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple=true)]
```

The previous attribute passes `AttributeTargets.Method` as a normal parameter and uses a named parameter to set the `AllowMultiple` property to `true`. Typically, all required attribute information is passed in a normal parameter while optional information is passed by named parameter. This way, the client is forced to pass all required information but can also choose which optional named parameters to include. Named parameters should always have a default value that will be used if a value is not explicitly specified by the client.

As previously mentioned, attributes should immediately precede the elements to which they are attached. Multiple attributes can be attached to a single entity by designating them individually or as a comma-delimited list. For instance, attributes can be defined individually like this:

```
[Serializable]
[Obsolete]
```

Or together like this:

```
[Serializable, Obsolete]
```

Another useful attribute worth noting is the `[Conditional()]` attribute defined in the `System.Diagnostics` namespace. This attribute causes the compiler to conditionally compile methods to which it is attached. Similar to the `#if` preprocessor directive, the `[Conditional()]` attribute indicates that a method should be compiled only if the specified value is defined. For example, the following method will only be compiled if the `DEBUG` value has been defined using the `#define` directive.

```
[System.Diagnostics.Conditional("DEBUG")]
private void log(string message)
{
  Console.WriteLine(message);
}
```

To log messages when `DEBUG` or `INFO` values have been defined, use multiple `[Conditional()]` attributes like this:

```
using System.Diagnostics; //declared at top of source file

[Conditional("DEBUG"), Conditional("INFO")]
private void log(string message)
{
  Console.WriteLine(message);
}
```

Though the `[Conditional()]` attribute functions similarly to the `#if` preprocessor directive, there are some key differences. First, unlike the `#if` directive, when the compiler encounters a `[Conditional()]` attribute attached to a method, it suppresses compilation of that method as well as any code that invokes the method. Since it removes all references to the conditional method, the `[Conditional()]` attribute can only be attached to methods that have a return value of `void`. On the other hand, unlike the `[Conditional()]` attribute, the `#if` directive is not restricted to methods. It can be used arbitrarily on any block of code. However, if compilation of a method is suppressed using the `#if` directive, compilation of all lines that call the conditional method must also be explicitly suppressed in a like manner. Lastly, the `#if` directive supports simple logical expressions while the `[Conditional()]` attribute does not.

Though the entity to which an attribute applies is usually clear, there are times when the intended relationship can be ambiguous. For example, an attribute that applies to an application's assembly is typically placed at the top of the source file, following any `using` statements, prior to the beginning of a class definition. However, a class attribute would be located at the same place. How does the compiler know if this attribute refers to the assembly or the class? The answer is that the target entity must be explicitly declared. For instance, an attribute immediately preceding a class definition, by default, applies to the class. If, however, the attribute is meant to apply to the assembly instead, the relationship must be explicitly stated as follows:

```
[assembly: Obsolete]
public class MathStuff
{
  //class implementation...
}
```

The previous attribute uses the `assembly` type identifier to indicate that the entire assembly, not just the `MathStuff` class, should be marked as obsolete. Table 5.1 documents the valid attribute type identifiers.

Table 5.1 ***Attribute type identifier (C#)***

Type Identifier	Description
`assembly`	Attribute refers to the assembly.
`event`	Attribute refers to the following event.
`field`	Attribute refers to the following field.
`method`	Attribute refers to the following method.
`module`	Attribute refers to the module.
`param`	Attribute refers to the following parameter.
`property`	Attribute refers to the following property.
`return`	Attribute refers to the return type of the following method.
`type`	Attribute refers to the following class or struct.

Table 5.2 presents a brief list of common attributes provided by the .NET framework. Of course, this list is far from exhaustive. See the .NET framework documentation for a complete list of available attributes.

Table 5.2 ***Common attributes (C#)***

Attribute	Attribute Class	Description
`[AssemblyTitle]`	`System.Reflection.AssemblyTitleAttribute`	Specifies the title, or name, of the assembly.
`[AssemblyVersion]`	`System.Relection.AssemblyVersionAttribute`	Specifies the assembly version number.
`[AttributeUsage]`	`System.AttributeUsageAttribute`	Specifies how an attribute

		class can be used.
[Conditional]	System.Diagnostics.ConditionalAttribute	Specifies a conditional method.
[NonSerialized]	System.NonSerializedAttribute	Specifies that an entity should not be serialized (similar to Java's transient statement).
[Obsolete]	System.ObsoleteAttribute	Specifies that an entity is obsolete.
[Serializable]	System.SerializableAttribute	Specifies an entity as serializable.
[WebMethod]	System.Web.Services.WebMethodAttribute	Specifies that a method is accessible using Web Services.
[WebService]	System.Web.Services.WebServiceAttribute	Specifies that a class should be exposed as a Web Service.

Creating Custom Attributes

Custom attributes are created by extending the System.Attribute class and adding the constructors and properties required to capture and store the appropriate custom fields. In addition, the proper usage of the attribute can be defined by attaching the [AttributeUsage()] attribute to the custom attribute class. The [AttributeUsage()] attribute defines one required parameter and two optional named parameters. The required parameter indicates the entity or entities to which the attribute can be applied. As shown in Table 5.3, valid entities are defined by the System.AttributeTargets enumeration.

Table 5.3 ***Attribute targets defined by System.AttributeTargets (C#)***

Name	Description
All	Attribute can be assigned to any application entity.
Assembly	Attribute can be assigned to an assembly.
Class	Attribute can be assigned to a class.
Constructor	Attribute can be assigned to a constructor.
Delegate	Attribute can be assigned to a delegate.
Enum	Attribute can be assigned to an enumeration.
Event	Attribute can be assigned to an event.
Field	Attribute can be assigned to a field.
Interface	Attribute can be assigned to an interface.
Method	Attribute can be assigned to a method.
Module	Attribute can be assigned to a module.
Parameter	Attribute can be assigned to a parameter.
Property	Attribute can be assigned to a property.
ReturnValue	Attribute can be assigned to a return value.
Struct	Attribute can be assigned to a struct.

Additionally, the AllowMultiple and Inherited named parameters indicate whether multiple attributes can be assigned to a single entity and whether the attribute is inherited by subclasses,

respectively. For example, the following attribute indicates that the associated custom attribute class can be applied to a class entity multiple times:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
```

In addition, multiple attribute targets can be specified using the bitwise logical OR operator ("|") like this:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
AllowMultiple=false)]
```

The previous attribute indicates that the associated custom attribute class can be applied to a class or struct entity only once.

To illustrate, imagine that we wanted to build an attribute that would track code changes made within a class. Important information to track might include the initials of the programmer that made the changes, the date the changes were made, a description of the changes, and an optional note field for any extraneous information. The `VersionControlAttribute` custom attribute presented in Listing 4.17 supports these requirements.

Listing 4.17 ***VersionControlAttribute custom attribute (C#)***

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class VersionControlAttribute: System.Attribute
{
  private string initials;
  private string date;
  private string description;
  private string note = ""; //optional, so assign default value


  public VersionControlAttribute(string initials, string date,
    string description)
  {
    this.initials = initials;
    this.date = date;
    this.description = description;
  }


  public string Note
  {
    get
    {
      return note;
    }

    set
    {
      note = value;
    }
  }


  public string Initials
  {
    get
    {
```

```
      return initials;
    }
  }


  public string Date
  {
    get
    {
      return date;
    }
  }


  public string Description
  {
    get
    {
      return description;
    }
  }
}
```

Notice from Listing 4.17 that the initials, date, and description values were made mandatory by placing them in the constructor while the note was exposed as an optional property. Also notice that the class is declared public. All custom attribute classes must be declared public.

When building custom attributes, constructors are used to capture required information while properties are provided for optional settings. Multiple overloaded constructors can be defined for a single custom attribute. When a custom attribute is used, the constructor values must be included first followed by any named parameters. For instance, given the custom attribute shown in Listing 4.17, the following attribute usage sets the value of the `Note` property to "For internal use only." and passes the string values "DRC", "3/18/2003", and "Created." to the custom attribute's constructor:

```
[VersionControl("DRC", "3/18/2003", "Created.",
  Note="For internal use only.")]
```

Note that the name of the named parameter must match the name of the property defined by the custom attribute. Of course, since the named parameter is optional, the attribute could have been used without it, like this:

```
[VersionControl("DRC", "3/18/2003", "Created.")]
```

Listing 4.18 demonstrates how the `VersionControlAttribute` custom attribute can be used to track code changes.

Listing 4.18 ***Class that uses the VersionControlAttribute custom attribute (C#)***

```
using System;

[VersionControl("DRC", "3/18/2003", "Created.",
  Note="For internal use only.")]
[VersionControl("DRC", "3/21/2003", "Added Multiply method.")]
[VersionControl("DRC", "3/22/2003", "Added Divide method.")]
public class MathStuff
{
  public int Add(int x, int y)
```

```
  {
    return x + y;
  }


  public int Subtract(int x, int y)
  {
    return x - y;
  }


  public int Multiply(int x, int y)
  {
    return x * y;
  }


  public int Divide(int x, int y)
  {
    if (y == 0)
      throw new DivideByZeroException();

    return x / y;
  }
}
```

Listing 4.18 uses a custom attribute to store version control information within the class's metadata. That's pretty neat but it doesn't do us any good unless we can extract the information. Listing 4.19 demonstrates how to read custom metadata information using a process called reflection.

Listing 4.19 ***Reads and displays VersionControlAttribute information (C#)***

```
using System;

public class VersionControlAttributeViewer
{
  public static void Main()
  {
    System.Reflection.MemberInfo member = typeof(MathStuff);

    VersionControlAttribute[] changes =
      (VersionControlAttribute[])member.GetCustomAttributes(
      typeof(VersionControlAttribute), false);

    foreach (VersionControlAttribute change in changes)
    {
      Console.WriteLine("Initials: {0}", change.Initials);
      Console.WriteLine("Date: {0}", change.Date);
      Console.WriteLine("Description: {0}", change.Description);

      if (change.Note != "")
        Console.WriteLine("Note: {0}", change.Note);

      Console.WriteLine();
    }
  }
}
```

Listing 4.19 begins by retrieving the type object associated with the `MathStuff` class from Listing 4.18. Next, using the `GetCustomAttributes()` method, it retrieves all attributes of the type `VersionControlAttribute` contained in the `MathStuff` class. This method can also be used without any type information to retrieve all custom attributes. Lastly, the code iterates through each attribute and writes the attribute information to the console.

This concludes our discussion on attributes. As we have seen, attributes are a flexible and powerful way to store custom metadata information within a compiled class. This information can then be used by compilers, integrated development environments, or any other applications capable of reading and interpreting this metadata.

Pointers (C#)

Pointers are variables that allow direct access to memory. Though common in C and C++ programming, pointers are rarely used in C# because they are complex, unsafe, and difficult to debug. Rather than pointers, Java and C# use references to refer to objects in memory. A reference can be thought of as a "safe pointer" because references overcome many of the most common problems associated with pointers including the following:

1. References must be initialized before use. This rule ensures that a reference always points to a valid object. Pointers, on the other hand, can be accessed without being initialized, resulting in unpredictable behavior.
2. References are type-safe. That is, they always point to the correct type of object. In contrast, regardless of how it was declared, a pointer can point to any memory address that may contain any type of object.
3. References to arrays are bounds checked to ensure that the specified array element exists. In contrast, with pointers it is possible (and very common) to read or write past the end of an array.
4. Unlike pointers, references cannot directly access memory and, therefore, cannot inadvertently or maliciously write to incorrect memory locations.
5. Resources accessed through references are managed by the runtime environment and are automatically garbage collected when no longer referenced. In contrast, memory leaks are a common problem with programs that employ pointers because their resources must be released manually.

At this point, you may be wondering why C# supports pointers at all. Though it is often better to avoid them, there are a few situations where pointers are desirable or necessary. Typically, these situations involve writing performance-critical algorithms or interoperating with unmanaged code that accepts or returns pointer types. However, keep in mind that many of the problems commonly associated with pointers are very difficult to debug and can result in unstable applications. Acknowledging this fact, C# requires any use of pointers to be explicitly declared as `unsafe` in the code. The `unsafe` keyword can be used to declare an entire class or struct as unsafe, like this:

```
unsafe class UnsafeClass
{
  //pointers can be used anywhere within this class
}

unsafe struct UnsafeStruct
{
  //pointers can be used anywhere within this class
```

```
}
```

Or a single method like this:

```
unsafe int UnsafeMethod()
{
  //pointers can be used anywhere within this method
}
```

Or a designated block of code like this:

```
unsafe
{
  //pointers can be used anywhere within this block
}
```

Or an instance variable like this:

```
class UnsafeClass
{
  public unsafe int* point;
}
```

Though methods and instance variables can be marked `unsafe`, local method variables cannot. Rather, local variables must be declared within an `unsafe` method or `unsafe` block.

Unlike references, pointers are restricted to value types like `int`, `long`, `float`, `double`, and `struct`. Pointers cannot point to reference types such as strings or classes. This rule is strictly enforced by the compiler. In addition, there is an important caveat to remember when using `struct` pointers. Since pointers do not support reference types, a pointer cannot point to a `struct` that contains any reference types. On the other hand, pointers can point to value types that are contained within a reference type (e.g., a public `int` field exposed by a class).

Put simply, a pointer is a variable that contains the address of another variable. Special characters are used to declare and reference pointers. Let's start by looking at how pointers are declared. Similar to C and C++, C# pointers are declared using the `*` operator. For example, the following code declares a pointer to an integer:

```
int* point;
```

Unlike C and C++, the `*` operator in C# is attached to the type rather than the variable. For example, in C and C++, the following code declares a single integer pointer and two regular integers:

```
int *point1, point2, point3; //declaration in C/C++
```

To declare three integer pointers in C and C++, they would have to be declared like this:

```
int *point1, *point2, *point3; //declaration in C/C++
```

In contrast, multiple integer pointers are declared like this in C#:

```
int* point1, point2, point3;
```

In addition to declaring pointers, the `*` operator (known as the *dereferencing* or *indirection* operator) is used to dereference them. Remember, a pointer is a variable that contains the address

of another variable. Therefore, whenever a pointer variable is referenced, it simply returns the address that it contains. Typically, however, we are more interested in the object stored at that address rather than the address itself. When attached to a pointer variable, the `*` operator "dereferences" the pointer and returns the object to which the pointer is pointing, rather than the address stored in the pointer.

Commonly used in conjunction with pointers, the `&` operator returns the address of an object. This address can then be assigned to a pointer of the object's type. To illustrate, consider the following code:

```
int x = 10; //declare integer x and assign a value of 10
int y = 0;  //declare integer y and assign a value of 0

int* px;    //declare px as a pointer to an integer
int* px2;   //declare px2 as a pointer to an integer

px = &x;    //assign address of x to px so that it points to x
px2 = px;   //assign px to px2 so that px2 also points to x

y = *px2;   //dereference px2, assign value to y (y is now 10)
```

The previous code declares two integers (`x` and `y`) and two pointers to integer (`px` and `px2`). The pointer `px` is then assigned the address of the integer `x` so that it points to `x`. Any reference to the variable `px` returns the address of integer `x`. The address stored in `px` is assigned to `px2` so that it also points to the integer `x`. To return the value of `x` (rather than its address), the `*` operator must be used to dereference the pointer. The code concludes by dereferencing the pointer `px2` and assigning the value of `x` to the integer variable `y`.

Pointer Arithmetic

It is possible to add to or subtract from a pointer's address through a process called *pointer arithmetic*. Using a valid pointer and an offset value, the values of surrounding addresses can be examined. For example, consider the `PointerArithmetic` program presented in Listing 4.20. This program populates an integer array with 10 integers and uses pointer arithmetic to iterate through the array, writing each element's value and address to the console.

Listing 4.20 ***Iterates through an array using pointer arithmetic (C#)***

```
using System;

public class PointerArithmetic
{
  private const int ARRAY_SIZE = 10;

  public unsafe static void Main()
  {
    int[] x = new int[ARRAY_SIZE];

    //populate array with values 1 through 10
    for (int y = 0; y < ARRAY_SIZE; y++)
    {
      x[y] = y+1;
    }

    //iterate through array using pointer arithmetic
    fixed (int* px = &x[0])
    {
```

```
      for (int y = 0; y < ARRAY_SIZE; y++)
      {
        Console.WriteLine("x[{0}] = {1} at {2}", y, *(px + y),
          (int)(px + y));
      }
    }
  }
}
```

There are a few items worth noting about Listing 4.20. First, notice that the pointer's address is cast to an `int` type for display. Since pointers store integer addresses, a pointer can be cast to or from any integer type. Second, note how parentheses are used to increment the pointer before it is dereferenced (e.g., `*(px + y)`). In this way, the pointer's address is incremented rather than the value to which it points. Finally, while reviewing Listing 4.20, you may have noticed a keyword that you didn't recognize. The `fixed` keyword plays a very important role when working with pointers. This keyword will be presented in detail after our discussion on pointer arithmetic. Figure 4.1 displays the output generated by Listing 4.20.

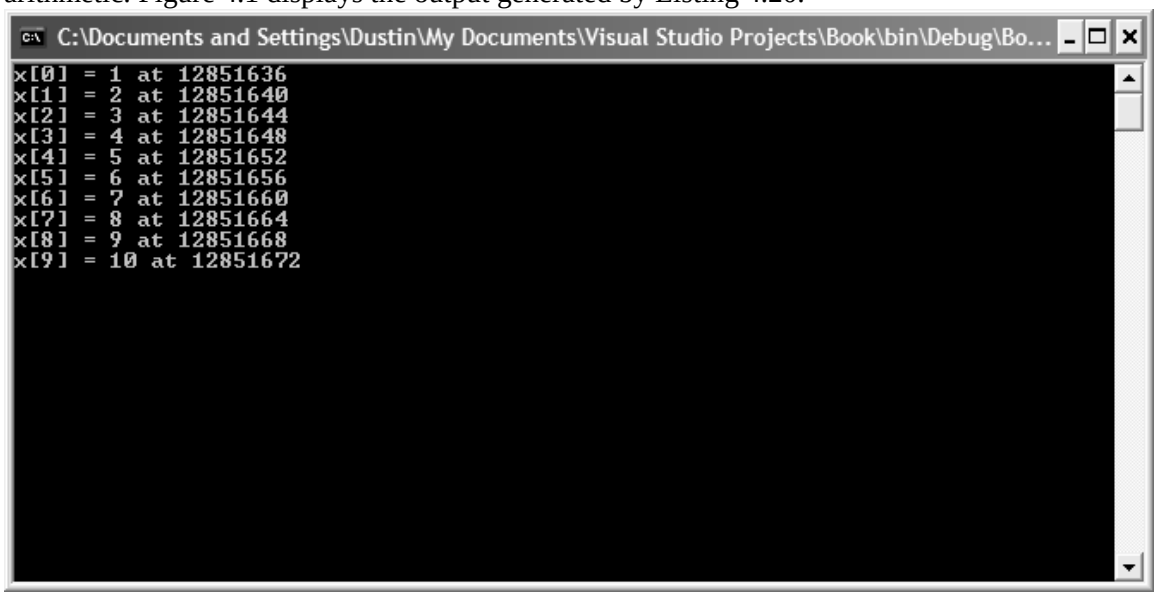

Figure 4.1 ***Output generated by Listing 4.20***

Carefully examine the addresses shown in Figure 4.1. Notice that the addresses are increasing by 4 bytes even though each iteration in Listing 4.20 incremented the pointer address by only one. This behavior is by design in order to simplify pointer arithmetic. Since an `int` occupies 4 bytes and `px` was declared as a pointer to `int`, the compiler automatically adds 4 bytes with each increment. Thus, adding 3 to an `int` pointer increments the address by 12. Likewise, since a `long` consumes 8 bytes, adding 3 to a `long` pointer increments the pointer's address by 24. You can determine the size of a value type using the `sizeof` operator like this:

```
int sizeOfInt = sizeof(int);
int sizeOfLong = sizeof(long);
int sizeOfFloat = sizeof(float);
int sizeOfDouble = sizeof(double);
```

`fixed` Keyword

As previously discussed, the .NET runtime provides automatic garbage collection. The garbage collector reduces memory fragmentation and improves memory access efficiency by moving objects around in memory. When an object is moved, the garbage collector automatically updates all references. However, the garbage collector has no knowledge of pointers. Of course, this behavior has the potential to wreak havoc with pointers since the objects they point to could be moved at any time. Fortunately, C# provides the `fixed` keyword to remedy this situation. Through a process known as *pinning*, the `fixed` keyword notifies the garbage collector that the object to which it points should not be moved. As demonstrated in Listing 4.20, the fixed keyword is used like this:

```
fixed (int* px = &x[0])
{
  //code that uses the px pointer
}
```

This code pins the integer array `x` for the duration of the `fixed` block. Since pinning has a detrimental effect on memory efficiency, objects should only be pinned when necessary and for the shortest duration possible. To ensure that a pointer's object is not moved out from under it, the compiler enforces the proper use of the `fixed` keyword.

To pin multiple objects within the same scope, simply stack the `fixed` statements like this:

```
fixed (int* px = &x[0])
fixed (int* px2 = &x[1])
{
  //code that uses the px and px2 pointers
}
```

The "stacked" fixed statements shown here are equivalent to the following:

```
fixed (int* px = &x[0])
{
  fixed (int* px2 = &x[1])
  {
    //code that uses the px and px2 pointers
  }
}
```

Traditionally, pointers have not been employed within a managed environment due to possibility of the garbage collector moving an object to which a pointer points. With the `fixed` keyword, the C# designers have overcome this limitation and provided a simple solution to this complex problem.

void Pointers

If you do not wish to declare the type to which a pointer will point, a generic pointer capable of pointing to any value type can be declared as follows:

```
void* genericPointer = (void*)pointerToInt;
void* genericPointer = (void*)pointerToLong;
```

As shown here, any pointer can be cast to a `void` pointer. Typically, `void` pointers are only used when calling legacy APIs that require them.

Pointers to Structs

As mentioned previously, a pointer can point to a `struct`. The caveat is that the `struct` cannot contain any reference types (since pointers cannot point to reference types). Though pointers to `struct`s work just like pointers to other types, there is some additional syntax to learn when using them. For instance, `struct` members can be accessed by dereferencing a `struct` pointer and then accessing its member normally like this:

```
struct Point
{
  int X;
  int Y;
}

Point p = new Point();
Point* pp = &p;

int x = (*pp).X;
int y = (*pp).Y;
```

Adopting some syntax from C++, it is possible to dereference a `struct` pointer and simultaneously access one of its members using the `->` operator like this:

```
int x = pp->X;
int y = pp->Y;
```

Stack Allocation

Within an `unsafe` class, method, or block, it is possible to manually allocate a portion of memory from the stack. Stack allocated memory is fast and efficient because it does not have to be garbage collected. Rather, stack allocated variables are automatically released when the current block goes out of scope. Since they are not garbage collected, stack allocated variables need not be pinned (i.e., declared within a `fixed` block). A block of memory is allocated from the stack using the `stackalloc` keyword as follows:

```
int* stackArray = stackalloc int[10];
```

Notice how stack allocation is similar to declaring an array. The previous code allocates 40 blocks of stack space (10 `int` variables at 4 bytes a piece) and returns a pointer to the first element (i.e., start of the memory block). Note that, for performance reasons, the block of memory is not initialized. Elements of the stack array can be initialized via pointer arithmetic as shown here:

```
*stackArray = 1;       //initialize element 1
*(stackArray + 1) = 2; //initialize element 2
*(stackArray + 2) = 3; //initialize element 3
```

Though it works, this method of accessing stack allocated memory is a bit cumbersome. Fortunately, there is a more convenient way. In C#, it is possible to access a stack allocated memory pointer as if it was an array like this:

```
stackArray[0] = 1; //initialize element 1
stackArray[1] = 2; //initialize element 2
```

```
stackArray[2] = 3; //initialize element 3
```

Keep in mind that stack allocated arrays are typically reserved for times when performance is absolutely critical. When using these arrays, it is easy to accidentally corrupt memory when using stack allocated arrays. This is because, unlike managed arrays, stack allocated arrays are not bounds checked. For instance, the following code will not generate an exception:

```
int* stackArray = stackalloc int[10];
stackArray[20] = 100;
```

Notice that the previous code accesses an element well beyond the allocated block of memory. Setting a value outside of the allocated stack space can produce unpredictable results. For instance, the value of a critical variable could be altered causing the program to crash or return invalid results. For this reason, stack allocated memory should be used carefully and only when performance requirements demand it.

Summary

This chapter presented several common programming structures. A thorough understanding of these basic structures is critical for understanding and fully exploiting the Java and C# languages. These structures were described as follows:

- In object-oriented programming, a *property* is an intrinsic characteristic of an object.
- When working with a class that represents a collection of objects, it may be convenient to access the class as if it were an array. C# supports this behavior through a feature known as an *indexer*.
- A *callback* is a function or method that a program indicates should be called when a certain event takes place (such as a user pressing a key on the keyboard or clicking a mouse button).
- Similar to a C/C++ function pointer, a `delegate` in C# provides a reference, or an alias, to a particular method.
- Included in a compilation as part of a class's metadata, an *attribute* is an annotation that is attached to a particular entity (e.g., class, field, property, method).
- *Pointers* are variables that allow direct access to memory. Though common in C and C++ programming, pointers are rarely used in C# because they are complex, unsafe, and difficult to debug.

Though they are implemented very differently, Java and C# both support properties and events. In addition, C# introduces several structures not currently found in Java including indexers, attributes, and pointers. Given the Java designers' disdain for pointers, it is doubtful that Java will ever support them. However, it is possible that Java may one day return the favor and adopt some of C#'s most popular features including indexers and attributes. In fact, work is already under way to add attribute-like functionality to Java.

This chapter concludes the narrative half of the book. The second half is presented in a reference format. In the next chapter, we will present many of the most common tools used when building Java and .NET solutions including utilities for compiling, debugging, documenting, packaging, and disassembling.