Chapter 19

Delegates and Events

Delegates and Events

Objectives

After completing this unit you will be able to:

- **Use delegate objects to implement callbacks.**
- **Use aggregations of delegate objects.**
- **Use delegate objects to implement and handle event notifications.**

Overview of Delegates and Events

- **In the previous two chapters we examined interfaces in some detail.**
- **One feature of interfaces is that they facilitate writing code in such a way that your program is *called into* by some other code.**
- **This style of programming has been available for a long time, under the guise of "callback" functions.**
- **In this chapter we examine *delegates* in C#, which can be thought of as type-safe and object-oriented callback functions.**
- **Delegates are the foundation for a more elaborate callback protocol, known as *events*.**
- **Events are a cornerstone of COM, the predecessor of .NET, and are widely used in Windows programming.**
- **We will study events and look at several example programs that illustrate delegates and events.**

Callbacks and Delegates

- **A callback function is one that your program specifies and "registers" in some way, and then gets called by another program.**
  - In C and C++ callback functions are implemented using function pointers.
- **In C# you can encapsulate a reference to a method inside a delegate object.**
- **While a function pointer can reference only a static function, a delegate can refer to either a static method or an instance method.**
- **When a delegate refers to an instance method, it stores both an object instance and an entry point to the instance method.**
  - The instance method can then be called through this object instance.
  - When a delegate object refers to a static method, it stores just the entry point of this static method.

Usage of Delegates

- **You can then pass this delegate object to other code, which can then call your method.**
  - The code that calls your delegate method does not have to know at compile time which method is being called; it only has to know the exact signature.
- **In C# a delegate is considered a reference type that is similar to a class type.**
  - A new delegate instance is created just like any other class instance, using the **new** operator.
  - C# delegates are implemented by the .NET Framework class library as a class, derived ultimately from **System.Delegate**.
- **Delegates are object-oriented and type-safe, and they enjoy the safety of the managed code execution environment.**

Declaring a Delegate

- **You declare a delegate in C# using a special notation with the keyword *delegate* and the signature of the encapsulated method.**
  - A suggested naming convention is to end your name with "Callback." Here is an example of a delegate declaration:

```
public delegate void NotifyCallback(decimal bal);
```

  - The **NotifyCallback** delegate in this example can contain a reference to any function with the return type **void**, and one parameter of type **decimal**.
  - A delegate reference is not restricted to any one class as long as the signature of the function matches.

Defining a Method

- **When you instantiate a delegate, you will need to specify a method, which must match the signature in the delegate declaration.**
- **The method may be either a static method or an instance method.**
- **Here are some examples of methods that can be hooked to the *NotifyCallback* delegate:**

```
private static void NotifyCustomer(decimal balance)
{
    Console.WriteLine("Dear customer,");
    Console.WriteLine(
       "   Account overdrawn, balance = {0}",
       balance);
}
private static void NotifyBank(decimal balance)
{
   Console.WriteLine("Dear bank,");
   Console.WriteLine(
      "   Account overdrawn, balance = {0}",
      balance);
}
private void NotifyInstance(decimal balance)
{
   Console.WriteLine("Dear instance,");
   Console.WriteLine(
      "   Account overdrawn, balance = {0}",
      balance);
}
```

Creating a Delegate Object

- **You instantiate a delegate object with the *new* operator, just as you would with any other class.**
  - The following code illustrates creating two delegate objects.
  - The first one is hooked to a static method, and the second to an instance method.
  - The second delegate object internally will store both a method entry point and an object instance that is used for invoking the method.

```
NotifyCallback custDlg = new
NotifyCallback(NotifyCustomer);
...
DelegateAccount da = new DelegateAccount();
NotifyCallback instDlg =
   new NotifyCallback(da.NotifyInstance);
```

Calling a Delegate

- **You "call" a delegate just as you would a method.**
  - The delegate object is not a method, but it has an encapsulated method.
  - The delegate object "delegates" the call to this encapsulated method, hence the name "delegate."
- **In the following code the delegate object *notifyDlg* is called whenever a negative balance occurs on a withdrawal.**
  - In this example the **notifyDlg** delegate object is initialized in the method **SetDelegate**.

```
private NotifyCallback notifyDlg;
...
public void SetDelegate(NotifyCallback dlg)
{
   notifyDlg = dlg;
}
...
public void Withdraw(decimal amount)
{
   balance -= amount;
   if (balance < 0)
      notifyDlg(balance);
}
```

Combining Delegate Objects

- **A powerful feature of delegates is that you can combine them, implementing an invocation list of methods.**
  - When such a delegate is called, all the methods on the invocation list will be called in turn.
  - The + operator can be used to combine the invocation methods of two delegate objects, and the – operator can be used to remove methods.

```
NotifyCallback custDlg = new
NotifyCallback(NotifyCustomer);
NotifyCallback bankDlg = new
NotifyCallback(NotifyBank);
NotifyCallback currDlg = custDlg + bankDlg;
// or NotifyCallback currDlg += bankDlg;
```

- **In this example we construct two delegate objects, each with an associated method.**
- **We then create a new delegate object whose invocation list will consist of both the methods *NotifyCustomer* and *NotifyBank*.**
  - When **currDlg** is called, these two methods will be invoked. Later on in the code we may remove a method.

```
currDlg -= bankDlg;
```

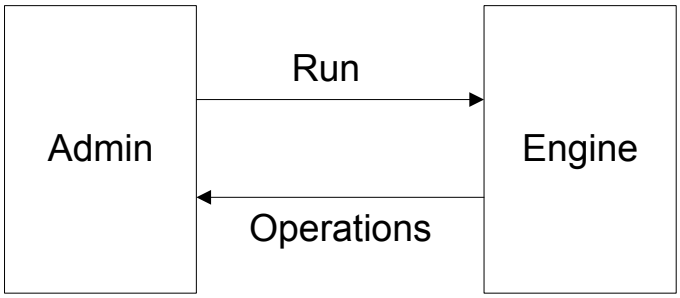
- **Now *NotifyBank* has been removed from the delegate, and the next time *currDlg* is called, only *NotifyCustomer* will be invoked.**

Complete Example

- **The program *DelegateAccount* illustrates using delegates in our bank account scenario.**
- **The file DelegateAccount.cs declares the delegate NotifyCallback.**
  - The class **DelegateAccount** contains methods matching the signature of the delegate.
  - The **Main** method instantiates delegate objects and combines them in various ways.
  - The delegate objects are passed to the **Account** class, which uses its encapsulated delegate object to invoke suitable notifications when the account is overdrawn.
  - Observe how dynamic and loosely coupled is this structure.
- **The *Account* class does not know or care which notification methods will be invoked in the case of an overdraft.**
- **It simply calls the delegate, which in turn calls all the methods on its invocation list.**
  - These methods can be adjusted at runtime.
- **Please examine the code online.**

Stock Market Simulation

- **As a further illustration of the use of delegates, consider the simple stock market simulation, implemented in the directory *StockMarket*.**
- **The simulation consists of two modules:**
  - The **Admin** module provides a user interface for configuring and running the simulation. It also implements operations called by the simulation engine.
  - The **Engine** module is the simulation engine. It maintains an internal clock and invokes randomly generated operations, based on the configuration parameters passed to it.
- **The figure shows the high level architecture of the simulation.**



Stock Market Simulation (Cont'd)

- **The following operations are available:**
  - PrintTick: shows each clock tick.
  - PrintTrade: shows each trade.
- **The following configuration parameters can be specified:**
  - Ticks on/off
  - Trades on/off
  - Count of how many ticks to run the simulation

Running the Simulation

- **Build and run the example program in *StockMarket*.**
- **Start with the default configuration: Ticks are OFF, Trades are ON, Run count is 100. (Note that the results are random and will be different each time you run the program.)**
- **The available commands are listed when you type "help" at the colon prompt. The commands are:**

```
count    set run count
ticks    toggle ticks
trades   toggle trades
config   show configuration
run      run the simulation
quit     exit the program
```

- **The output shows clock tick, stock, price, volume.**

```
: run
   6  IBM    112   400
  24  MSFT    60   600
  38  MSFT    66   800
  43  MSFT    60   500
  58  MSFT    66   600
  59  INTC    91   800
  60  IBM    107   600
  64  MSFT    72   900
  76  IBM    102   800
  83  IBM     97   900
  86  MSFT    79   500
  94  MSFT    86   100
  97  INTC    81   500
  99  MSFT    94   900
 100  MSFT    85   800
```

Delegate Code

- **Two delegates are declared in the *Admin.cs* file.**

```
public delegate void TickCallback(int ticks);
public delegate void TradeCallback(int ticks,
   string stock, int price, int volume);
```

- **As we saw in the previous section, a delegate is similar to a class, and a delegate object is instantiated by *new*.**

```
TickCallback tickDlg = new TickCallback(PrintTick);
TradeCallback tradeDlg = new
TradeCallback(PrintTrade);
```

- **A method is passed as the parameter to the delegate constructor. The method signature must match that of the delegate.**

```
public static void PrintTick(int ticks)
{
   Console.Write("{0} ", ticks);
   if (++printcount == LINECOUNT)
   {
      Console.WriteLine();
      printcount = 0;
   }
}
```

Passing the Delegates to the Engine

- **The *Admin* class passes the delegates to the *Engine* class in the constructor of the *Engine* class.**

```
Engine engine = new Engine(tickDlg, tradeDlg);
```

- **The heart of the simulation is the *Run* method of the *Engine* class.**
  - At the core of the **Run** method is assigning simulated data based on random numbers.
- **We use the *System.Random* class, which we discussed in Chapter 12.**

```
double r = rangen.NextDouble();
if (r < tradeProb[i])
{
   int delta = (int) (price[i] * volatility[i]);
   if (rangen.NextDouble() < .5)
   {
      delta = -delta;
   }
   price[i] += delta;
   int volume=rangen.Next(minVolume,maxVolume)*100;
   tradeOp(tick, stocks[i], price[i], volume);
```

Using the Delegates

- **In the Engine class, delegate references are declared:**

```
TickCallback tickOp;
TradeCallback tradeOp;
```

- **The delegate references are initialized in the Engine constructor:**

```
public Engine(TickCallback tickOp,
        TradeCallback tradeOp)
{
   this.tickOp = tickOp;
   this.tradeOp = tradeOp;
}
```

- **The method that is wrapped by the delegate object can then be called through the delegate reference:**

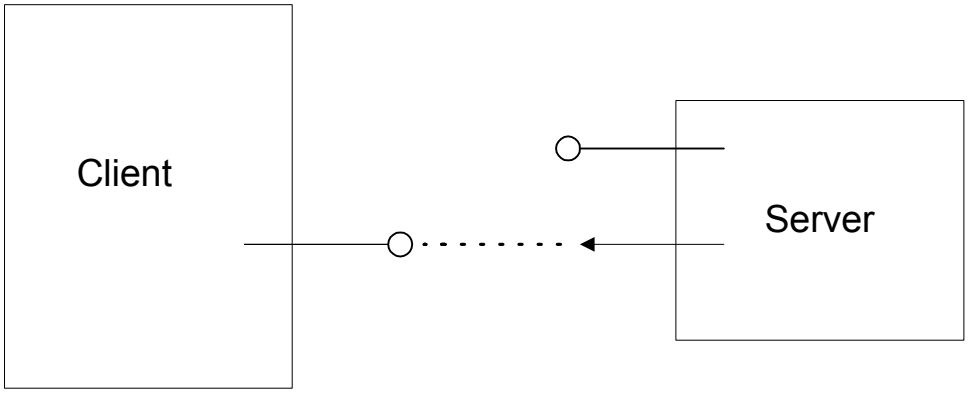
```
if (showTicks)
   tickOp(tick);
```

Events

- **Delegates are the foundation for a more elaborate callback protocol known as *events*.**
- **Conceptually, servers implement *incoming* interfaces, which are called by clients.**
  - In a diagram, such an interface may be shown with a small bubble (a notation used in COM).
- **Sometimes a client may wish to receive notifications from a server when certain "events" occur.**
- **In such a case the server will specify an *outgoing* interface.**
  - The server defines the interface and the client implements it.
  - In a diagram, such an interface may be shown with an arrow (again, a notation used in COM).

Events (Cont'd)

- The figure illustrates a server with one incoming interface and one outgoing interface.



- In the case of the outgoing interface, the client will implement an incoming interface, which the server will call.

Events in C# and .NET

- **The .NET Framework provides an easy-to-use implementation of the event paradigm built on delegates.**
- **C# simplifies working with .NET events by providing the keyword *event* and operators to hook up event handlers to events and to remove them.**
- **We will examine this event architecture through salient code from the example program *EventDemo*, which illustrates a chat room.**

Server-Side Event Code

- **We begin with server-side code, in *ChatServer.cs*.**
  - The .NET event architecture uses a specific signature:

```
public delegate void JoinHandler(object sender,
                                 ChatEventArg e);
```

  - The first parameter specifies the object that sent the event notification, the second parameter is used to pass data along with the notification.

- **Typically, you will derive a class from *EventArg* to hold your specific data.**

```
public class ChatEventArg : EventArgs
{
   public ChatEventArg(string name){...}
   ...
}
```

Delegate Objects

- **A delegate object reference is declared using the keyword *event*.**

```
public event JoinHandler Join;
```

- **A helper method is typically provided to facilitate calling the delegate object(s) that have been hooked up to the event.**

```
protected void OnJoin(ChatEventArg e)
{
   if (Join != null)
   {
      Join(this, e);
   }
}
```

- **A test for *null* is made in case no delegate objects have been hooked up to the event.**
- **Typically, access is specified as** protected **so that a derived class has access to this helper method.**
  - You can then "fire" the event by calling the helper method.

```
public void JoinChat(string name)
{
   members.Add(name);
   OnJoin(new ChatEventArg(name));
}
```

Client-Side Event Code

- **The client provides event handler functions.**

```
public static void OnJoinChat(object sender,
ChatEventArg e)
{
   Console.WriteLine(
      "sender = {0}, {1} has joined the chat",
      sender, e.Name);
}
```

- **The client hooks the handler to the event, using the += operator.**

```
ChatServer chat = new ChatServer("OI Chat Room");
//Register to get event notifications from server
chat.Join += new JoinHandler(OnJoinChat);
```

- **The event starts out as null, and event handlers get added through +=.**
  - All of the registered handlers will get invoked when the event delegate is called.
  - You may unregister a handler through `-=`.

Chat Room Example

- **The chat room example in *EventDemo* illustrates the complete architecture on both the server and client side.**
- **The server provides the following methods:**
  - JoinChat
  - QuitChat
  - ShowMembers
- **Whenever a new member joins or quits, the server sends a notification to the client.**
  - The event handlers print out an appropriate message.
- **Here is the output from running the program:**

```
sender = OI Chat Room, Michael has joined the chat
sender = OI Chat Room, Bob has joined the chat
sender = OI Chat Room, Sam has joined the chat
--- After 3 have joined---
Michael
Bob
Sam
sender = OI Chat Room, Bob has quit the chat
--- After 1 has quit---
Michael
Sam
```

Client Code

- **The client program provides event handlers.**
  - It instantiates a server object and then hooks up its event handlers to the events.
  - The client then calls methods on the server.
  - These calls will trigger the server firing events back to the client, which get handled by the event handlers.

Server Code

- **The server provides code to store in a collection the names of people who have joined the chat.**
- **When a person quits the chat, the name is removed from the collection.**
  - Joining and quitting the chat triggers firing an event back to the client.
  - The server also contains the "plumbing" code for setting up the events, including declaration of the delegates, the events, and the event arguments.
  - There are also helper methods for firing the events.
- **It may appear that there is a fair amount of such "plumbing" code, but it is much simpler than the previous connection point mechanism used by COM for events.**
  - Also, in practice various wizards and other tools will generate the infrastructure for you automatically.
- **Events are used extensively in Windows programming.**

Summary

- **Delegates can be thought of as type-safe and object-oriented callback functions.**
- **In C# you can encapsulate a reference to a method inside a delegate object.**
- **You can then pass this delegate object to other code, which can then call your method.**
- **The code that calls your delegate method does not have to know at compile time which method is being called.**
- **Delegates are the foundation for a more elaborate callback protocol, known as events.**