Delegates and Events

- A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure.

- A delegate allow us to specify what the function we'll be calling *looks like* without having to specify *which* function to call.

- Delegate declaration looks just like the declaration for a function, except that, **we're declaring the signature of functions that this delegate can reference.**

(Try to remember exactly even if you do not understand now.)

```
using System;

namespace Akadia.BasicDelegate
{
    // Declaration
    public delegate void SimpleDelegate();

    class TestDelegate
    {
        public static void MyFunc()
        {
            Console.WriteLine("I was called by delegate ...");
        }

        public static void Main()
        {
            // Instantiation
            SimpleDelegate simpleDelegate = new SimpleDelegate(MyFunc);

            // Invocation
            simpleDelegate();
        }
    }
}
```

```
class Program
    {
        public class MyClass
        {
            // Declare no return type.
            public delegate void LogHandler(string message);
            // Check if it is pointing to a function .
            public void Process(LogHandler logHandler)
            {
                if (logHandler != null)     logHandler("Process() begin");
                if (logHandler != null)     logHandler("Process() end");
            }
        }

        // Test Application to use the defined Delegate
        public class TestApplication
        {
            static void Logger(string s)
            {
                Console.WriteLine(s);
            }

            static void Main(string[] args)
            {
                MyClass myClass = new MyClass();

                // Crate an instance of the delegate.
                MyClass.LogHandler myLogger = new MyClass.LogHandler(Logger);
                myClass.Process(myLogger);
            }
        }
    }
```

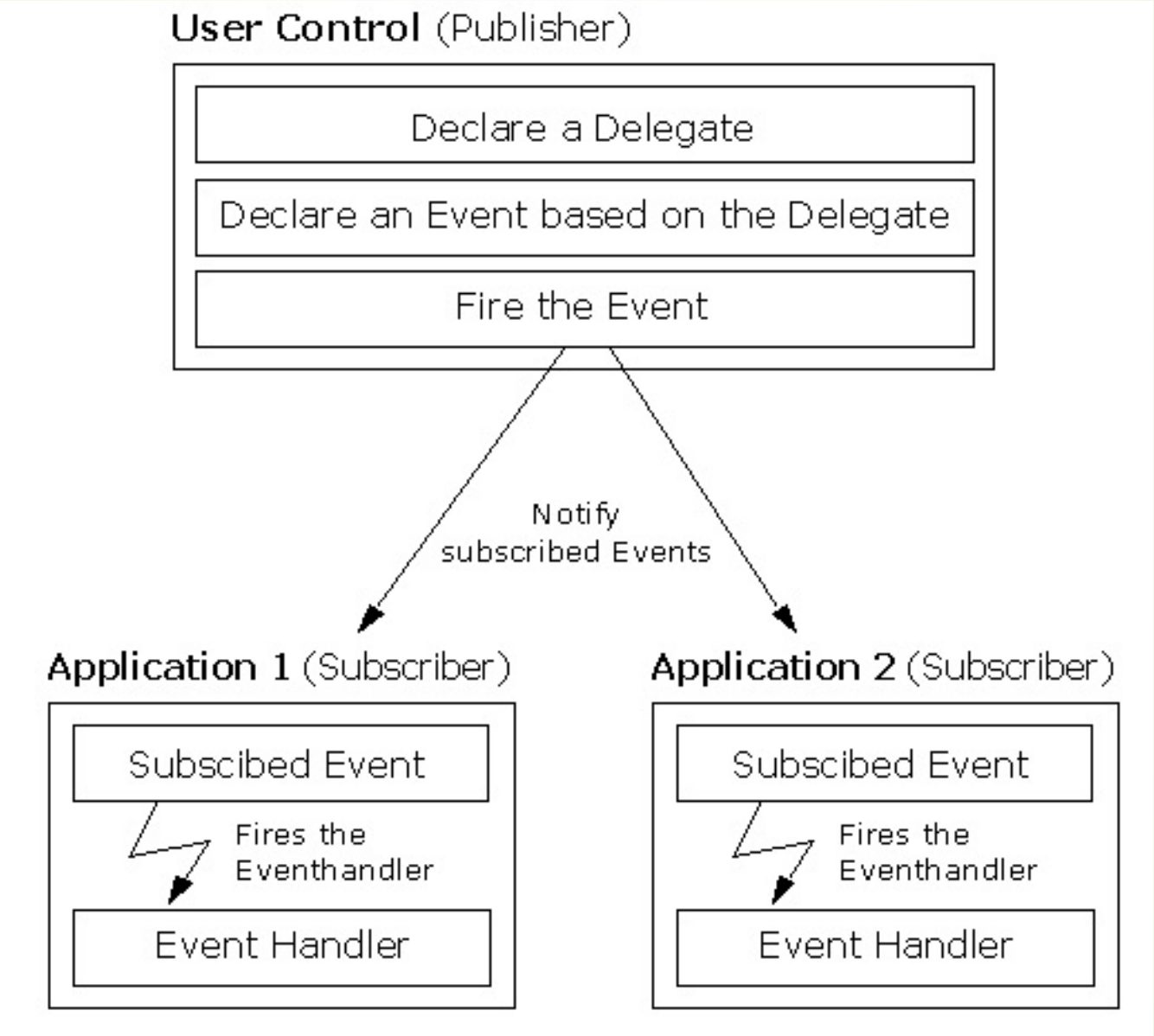
```
public class FileLogger
    {
        FileStream fileStream;
        StreamWriter streamWriter;
        // Constructor
        public FileLogger(string filename)
        {
            fileStream = new FileStream(filename, FileMode.Create);
            streamWriter = new StreamWriter(fileStream);
        }
        // Member Function which is used in the Delegate
        public void Logger(string s)
        {
            streamWriter.WriteLine(s);
        }
        public void Close()
        {
            streamWriter.Close();
            fileStream.Close();
        }
    }
    public class TestApplication
    {
        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");
            MyClass myClass = new MyClass();
            MyClass.LogHandler myLogger = new MyClass.LogHandler(fl.Logger);

            myClass.Process(myLogger);

fl.Close();
```

No change to Process() function; the code to all the delegate is the same regardless of whether it refers to a **static** or **member** function.

Events

- The Event model in C# has its roots in the event programming model that is popular in asynchronous programming.
- The foundation behind it is the idea of "publisher and subscribers."
- ***publishers*** will do some logic and publish an "event." They will then send out their event only to ***subscribers*** who have subscribed to receive the specific event.

- Any object can *publish* a set of events to which other applications can *subscribe*. When the publishing class raises an event, all the subscribed applications are notified.



Event Handlers

- Event Handlers in the .NET Framework return void and take two parameters.
  - **The first parameter is the source of the event**; that is the publishing object.
  - The second parameter is an object derived from EventArgs.
- **Events are properties** of the class publishing the event.
- Events can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the event.

```
public class Clock
    {
        private int _hour;        private int _minute;        private int _second;
        public delegate void SecondChangeHandler(object clock, TimeInfoEventArgs timeInformation);
        // The event we publish
        public event SecondChangeHandler SecondChange;
        // The method which fires the Event
        protected void OnSecondChange(object clock, TimeInfoEventArgs timeInformation)
        {
            // Check if there are any Subscribers
            if (SecondChange != null)
            {
                // Call the Event
                SecondChange(clock, timeInformation);
            }
        }
        // Set the clock running, it will raise an event for each new second
        public void Run()
        {
            for (; ; )
            {   Thread.Sleep(1000);
                // Get the current time
                System.DateTime dt = System.DateTime.Now;
                // If the second has changed notify the subscribers
                if (dt.Second != _second)
                {
                    // Create the TimeInfoEventArgs object to pass to the subscribers
               TimeInfoEventArgs timeInformation = new TimeInfoEventArgs(dt.Hour, dt.Minute, dt.Second);

                    // If anyone has subscribed, notify them
                    OnSecondChange(this, timeInformation);
                }
                // update the state
                _second = dt.Second;     _minute = dt.Minute;  _hour = dt.Hour;

            } } }
```

- The Clock class could simply print the time rather than raising an event.
- The advantage of the publish / subscribe is that any number of classes can be notified when an event is raised.
- The subscribing classes do not need to know how the Clock works,
- Clock does not need to know what they are going to do in response to the event.
- Similarly a button can publish an Onclick event, and any number of unrelated objects can subscribe to that event, receiving notification when the button is clicked.

- **The publisher and the subscribers are decoupled by the delegate.**
- This is highly desirable as it makes for more **flexible** and **robust** code.
- The clock can change how it detects time without breaking any of the subscribing classes.
- The subscribing classes can change how they respond to time changes without breaking the Clock.

Why use events for what I can do with Delegates?

- The rationale of using events instead of delegates is the same as for using properties instead of fields - *data encapsulation*. It's bad practice to expose fields (whatever they are - primitive fields or delegates) directly.