Advanced Programming C#

Lecture 8

dr inż. Małgorzata Janik

malgorzata.janik@pw.edu.pl

C#: events & delegates

Class members

- Constructors
- Destructors
- Fields
- Methods
- Properties
- Indexers
- **Delegates**
- **Events**
- Nested Classes

Delegates

- A delegate is a C# language element that allows you to reference a method.
- “Why do I need a reference to a method?”. The answer boils down to giving you maximum flexibility to implement any functionality you want at runtime.
- Think about how you use methods right now. You write an algorithm that does its thing by manipulating the values of variables and calling methods directly by name.
  - What if you wanted an algorithm that was very flexible, reusable, and allowed you to implement different functionality as the need arises?
  - Furthermore, let’s say that this was an algorithm that supported some type of data structure that you wanted to have sorted, but you also want to enable this data structure to hold different types. If you don’t know what the types are, how could you decide an appropriate comparison routine? Perhaps you could implement an if/then/else or switch statement to handle well-known types, but this would still be limiting and require overhead to determine the type.
  - You could solve this problem by passing a delegate to your algorithm and letting the contained method, which the delegate refers to, perform the comparison operation.

http://www.akadia.com/services/dotnet_delegates_and_events.html

Delegates

`delegate [result-type] identifier ([parameters]);`
where:
result-type: The result type, which matches the return type of the function.
identifier: The delegate name.
parameters: The Parameters, that the function takes.

Examples:

`public delegate void SimpleDelegate ()`

This declaration defines a delegate named `SimpleDelegate`, which will encapsulate any method that takes no parameters and returns no value.

`public delegate int ButtonClickHandler (object obj1, object obj2)`

This declaration defines a delegate named `ButtonClickHandler`, which will encapsulate any method that takes two objects as parameters and returns an int.

There are three steps in defining and using delegates:

- Declaration
- Instantiation
- Invocation

http://www.akadia.com/services/dotnet_delegates_and_events.html

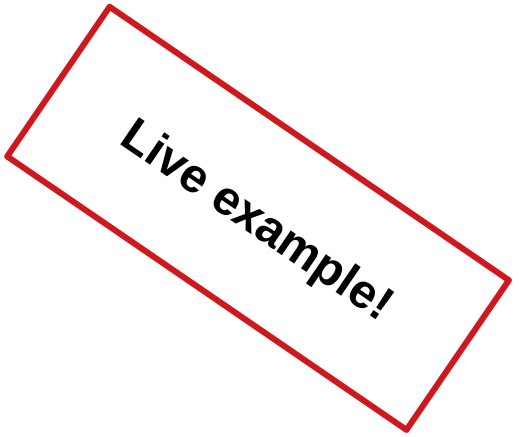
the very basic delegate

```
// 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();
  }
}
```



http://www.akadia.com/services/dotnet_delegates_and_events.html

calling static functions

```
// Delegate Specification
public class MyClass
{
    // Declare a delegate that takes a
    // single string parameter
    // and has no return type.
    public delegate void LogHandler(string message);

    // The use of the delegate is just like
    // calling a function directly,
    // though we need to add a check to see
    // if the delegate is null
    // (that is, not pointing to a function)
    // before calling the function.
    public void Process(LogHandler logHandler)
    {
        if (logHandler != null)
        {
            logHandler("Process() begin");
            logHandler ("Process() end");
        }
    }
}
```

```
// Test Application to use the defined Delegate
public class TestApplication
{

    // Static Function: To which is used in the Delegate. To call the Process()
    // function, we need to declare a logging function: Logger() that matches
    // the signature of the delegate.
    static void Logger(string s)
    {
        Console.WriteLine(s);
    }

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

        // Crate an instance of the delegate, pointing to the logging function.
        // This delegate will then be passed to the Process() function.
        MyClass.LogHandler myLogger = new MyClass.LogHandler(Logger);
        myClass.Process(myLogger);
    }
```

Static function

Output:

```
Process() begin
Process() end
```

http://www.akadia.com/services/dotnet_delegates_and_events.html

calling static functions

```
// Delegate Specification
public class MyClass
{
    // Declare a delegate that takes a
    // single string parameter
    // and has no return type.
    public delegate void LogHandler(string message);

    // The use of the delegate is just like
    // calling a function directly,
    // though we need to add a check to see
    // if the delegate is null
    // (that is, not pointing to a function)
    // before calling the function.
    public void Process(LogHandler logHandler)
    {
        if (logHandler != null)
        {
            logHandler("Process() begin");
            logHandler ("Process() end");
        }
    }
}
```

What if we want to put Logger function inside another class?

```
// Test Application to use the defined Delegate
public class TestApplication
{

    // Static Function: To which is used in the Delegate. To call the Process()
    // function, we need to declare a logging function: Logger() that matches
    // the signature of the delegate.
    static void Logger(string s)
    {
        Console.WriteLine(s);
    }

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

        // Crate an instance of the delegate, pointing to the logging function.
        // This delegate will then be passed to the Process() function.
        MyClass.LogHandler myLogger = new MyClass.LogHandler(Logger);
        myClass.Process(myLogger);
    }
```

Static function

Output:

```
Process() begin
Process() end
```

http://www.akadia.com/services/dotnet_delegates_and_events.html

Calling Member Functions

```
// Delegate Specification
public class MyClass
{
    // Declare a delegate that takes a
    // single string parameter
    // and has no return type.
    public delegate void LogHandler(string message);

    // The use of the delegate is just like
    // calling a function directly,
    // though we need to add a check to see
    // if the delegate is null
    // (that is, not pointing to a function)
    // before calling the function.
    public void Process(LogHandler logHandler)
    {
        if (logHandler != null)
        {
            logHandler("Process() begin");
            logHandler ("Process() end");
        }
    }
}
```

```
// The FileLogger class merely encapsulates the file I/O
    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); streamWriter.Flush();
        }

        public void Close()
        {
            streamWriter.Close();
            fileStream.Close();
        }
    }
```

Another class

```
// Main() is modified so that the delegate points to the Logger()
// function on the fl instance of a FileLogger. When this delegate
// is invoked from Process(), the member function is called and
// the string is logged to the appropriate file.
public class TestApplication
{
    static void Main(string[] args)
    {
        FileLogger fl = new FileLogger("process.log");

        MyClass myClass = new MyClass();

        // Crate an instance of the delegate, pointing to the Logger()
        // function on the fl instance of a FileLogger.
        MyClass.LogHandler myLogger = new MyClass.LogHandler(fl.Logger);
        myClass.Process(myLogger);
        fl.Close();
    }
}
```

http://www.akadia.com/services/dotnet_delegates_and_events.html

Multicasting

```
// Delegate Specification
public class MyClass
{
    // Declare a delegate that takes a
    // single string parameter
    // and has no return type.
    public delegate void LogHandler(string message);

    // The use of the delegate is just like
    // calling a function directly,
    public void Process(LogHandler logHandler)
    {
        if (logHandler != null)
        {
            logHandler("Process() begin");
            logHandler ("Process() end");
        }
    }
}
```

```
// The FileLogger class merely encapsulates the file I/O
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);  s
    }

    public void Close()
    {
        streamWriter.Close();
        fileStream.Close();
    }
}
```

Another class

```
// Test Application which calls both Delegates
public class TestApplication
{
    // Static Function which is used in the Delegate
    static void Logger(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        FileLogger fl = new FileLogger("process.log");

        MyClass myClass = new MyClass();

        // Crate an instance of the delegates, pointing to the static
        // Logger() function defined in the TestApplication class and
        // then to member function on the fl instance of a FileLogger.
        MyClass.LogHandler myLogger = null;
        myLogger += new MyClass.LogHandler(Logger);
        myLogger += new MyClass.LogHandler(fl.Logger);

        myClass.Process(myLogger);
        fl.Close();
    }
}
```

Static function

Adding both at the same time with + !

```
# SimpleDelegate4.exe
Process() begin
Process() end
# cat process.log
Process() begin
Process() end
```

http://www.akadia.com/services/dotnet_delegates_and_events.html

Multicasting

```
// Test Application which calls both Delegates
public class TestApplication
{
     // Static Function which is used in the Delegate
    static void Logger(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        FileLogger fl = new FileLogger("process.log");

        MyClass myClass = new MyClass();

        // Crate an instance of the delegates, pointing to the static
        // Logger() function defined in the TestApplication class and
        // then to member function on the fl instance of a FileLogger.
        MyClass.LogHandler myLogger = null;
        myLogger += new MyClass.LogHandler(Logger);
        myLogger += new MyClass.LogHandler(fl.Logger);

        myClass.Process(myLogger);

        MyLogger -= new MyClass.LogHandler(Logger);
        myLogger -= new MyClass.LogHandler(fl.Logger);

        fl.Close();
    }
}
```

Adding both at the same time with + !

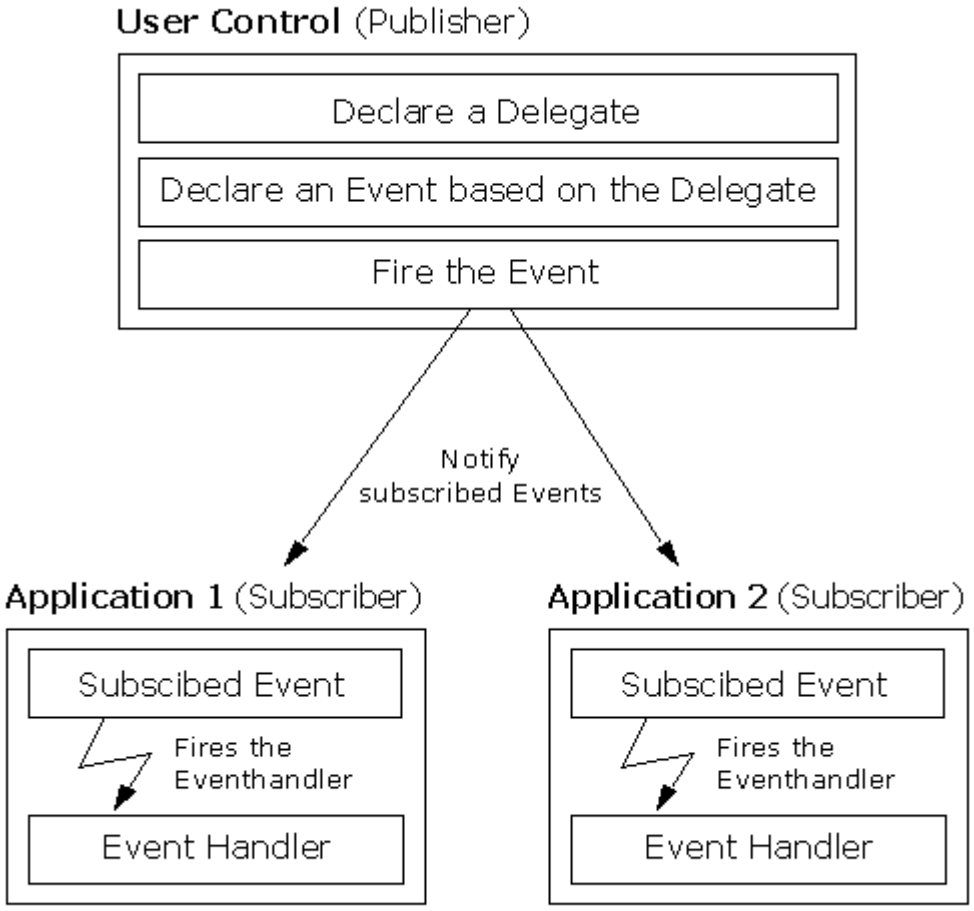
Subtracting with - !

http://www.akadia.com/services/dotnet_delegates_and_events.html

Events

The basic foundation behind event programming model is the idea of "publisher and subscribers." In this model, you have publishers who will do some logic and publish an "event." Publishers will then send out their event only to subscribers who have subscribed to receive the specific event.

In C#, 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.



Conventions

The following important conventions are used with events:

- **Event Handlers** in the .NET Framework **return void** and **take two parameters**.
- The **first paramter 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.
- The keyword ***event*** controls how the event property is accessed by the subscribing classes.

http://www.akadia.com/services/dotnet_delegates_and_events.html

Let's modify our logging example from above to use an event rather than a delegate

```
/* ========= Publisher of the Event ============== */
public class MyClass
{
    // Define a delegate named LogHandler,
    // which will encapsulate any method that takes
    // a string as the parameter and returns no value
    public delegate void LogHandler(string message);

    // Define an Event based on the above Delegate
    public event LogHandler Log;

// Instead of having the Process() function take a delegate
// as a parameter, we've declared a Log event. Call the Event,
// using the OnXXXX Method, where XXXX is the name of the Event.
    public void Process()
    {
        OnLog("Process() begin");
        OnLog("Process() end");
    }

    // By Default, create an OnXXXX Method, to call the Event
    protected void OnLog(string message)
    {
        if (Log != null)
        {
            Log(message);
        }
    }
}
```

```
/* ========= Subscriber of the Event ============== */
// It's now easier and cleaner to merely add instances
// of the delegate to the event, instead of having to
// manage things ourselves
public class TestApplication
{
    static void Logger(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        FileLogger fl = new FileLogger("process.log");
        MyClass myClass = new MyClass();

        // Subscribe the Functions Logger and fl.Logger
        myClass.Log += new MyClass.LogHandler(Logger);
        myClass.Log += new MyClass.LogHandler(fl.Logger);

        // The Event will now be triggered in the Process() Method
        myClass.Process();

        fl.Close();
    }
}
```

```
// The FileLogger class merely encapsulates the file I/O
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);  streamWriter.Flush();
    }

    public void Close()
    {
        streamWriter.Close();
        fileStream.Close();
    }
}
```

nd_events.html

```
/* ========= Publisher of the Event ============== */
public class MyClass
{
    // Define a delegate named LogHandler,
    // which will encapsulate any method that takes
    // a string as the parameter and returns no value
    public delegate void LogHandler(string message);

    // Define an Event based on the above Delegate
    public event LogHandler Log;

// Instead of having the Process() function take a delegate
// as a parameter, we've declared a Log event. Call the Event,
// using the OnXXXX Method, where XXXX is the name of the Event.
    public void Process()
    {
        OnLog("Process() begin");
        OnLog("Process() end");
    }

    // By Default, create an OnXXXX Method, to call the Event
    protected void OnLog(string message)
    {
        if (Log != null)
        {
            Log(message);
        }
    }
}
```

delegate that subscribers must implement

event we publish

Code that triggers OnXXX method (notify subscribers)

method which fires the event

Proper C# Convention

```
/* ======== Subscriber of the Event ============== */
blisher of the Event ============== */
Class
{
    // Define a delegate named LogHandler,
    // which will encapsulate any method that takes
    // a string as the parameter and returns no value
    public delegate void LogHandler(object o, MsgEventArgs msg);

    // Define an Event based on the above Delegate
    public event LogHandler Log;

// Instead of having the Process() function take a delegate
// as a parameter, we've declared a Log event. Call the Event,
// using the OnXXXX Method, where XXXX is the name of the Event.
) Method
    public void Process()
    {
        OnLog("Process() begin");
        OnLog("Process() end");
    }

    // By Default, create an OnXXXX Method, to call the Event
    protected void OnLog(string message)
    {
        if (Log != null)
        {
            MsgEventArgs event = new MsgEventArgs(message);
            Log(this, event);
        }
    }
}
```

delegate that subscribers must implement

event we publish

Code that triggers OnXXX method (notify subscribers)

method which fires the event

```
class MsgEventArgs : EventArgs{
    public readonly string msg;
    public MsgEventArgs(string m){
        msg=m;
    }
}
```

Clock

- **Clock**: a class that rises an event every second when started
- **DisplayClock**: subscribes to Clock's events. Displays current time.
- **LogClock**: subscribes to Clock's events.

  Write current time to file.
- **Test Application**: should create new Clock, one DisplayClock, one LogClock, subscribe DisplayClock and LogClock to Clock. Start clock.

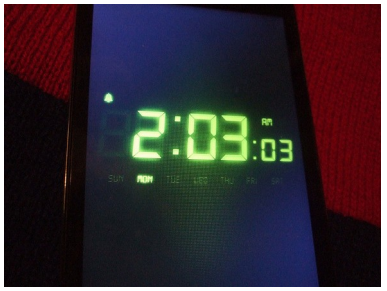
Clock

Clock

EVENT
hour, min, sec

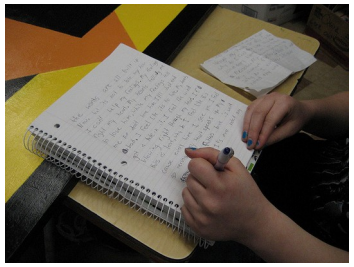
every
second

DisplayClock



EVENT
hour, min, sec

LogClock



any number of classes can be notified when an event is raised!

Test Application

```
/* ====================== Test Application ============================== */

// Test Application which implements the
// Clock Notifier - Subscriber Sample
public class Test
{
   public static void Main()
   {
      // Create a new clock
      Clock theClock = new Clock();

      // Create the display and tell it to
      // subscribe to the clock just created
      DisplayClock dc = new DisplayClock();
      dc.Subscribe(theClock);

      // Create a Log object and tell it
      // to subscribe to the clock
      LogClock lc = new LogClock();
      lc.Subscribe(theClock);

      // Get the clock started
      theClock.Run();
   }
}
```

Clock

DisplayClock
+ subscribe to events

LogClock
+ subscribe to events

http://www.akadia.com/services/dotnet_delegates_and_events.html

Contener sent with Event

```
// The class to hold the information about the event
  // in this case it will hold only information
  // available in the clock class, but could hold
  // additional state information
  public class TimeInfoEventArgs : EventArgs
  {
     public TimeInfoEventArgs(int hour, int minute, int second)
     {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
     }
     public readonly int hour;
     public readonly int minute;
     public readonly int second;
  }
```

EVENT
hour, min, sec

Events should derive from EventArgs class.

http://www.akadia.com/services/dotnet_delegates_and_events.html

Clock

```
/* ====================== Event Publisher ================================
*/

   // Our subject -- it is this class that other classes
   // will observe. This class publishes one event:
   // SecondChange. The observers subscribe to that event.
   public class Clock
   {
      // Private Fields holding the hour, minute and second
      private int _hour;
      private int _minute;
      private int _second;

      // The delegate named SecondChangeHandler, which will encapsulate
      // any method that takes a clock object and a TimeInfoEventArgs
      // object as the parameter and returns no value. It's the
      // delegate the subscribers must implement.
      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(
         TimeInfoEventArgs timeInformation
      )
      {
         // Check if there are any Subscribers
         if (SecondChange != null)
         {
            // Call the Event
            SecondChange(this,timeInformation);
         }
      }
```

delegate that subscribers must implement

event we publish

method which fires the event

```
// Set the clock running, it will raise an
      // event for each new second
      public void Run()
      {
         for(;;)
         {
            // Sleep 1 Second
            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 (timeInformation);
            }

            // update the state
            _second = dt.Second;
            _minute = dt.Minute;
            _hour = dt.Hour;

         }
      }
   }
```

notify subscribers

http://www.akadia.com/services/dotnet_delegates_and_events.html

DisplayClock

```
/* ======================= Event Subscriber 1 ============================== */

// An observer. DisplayClock subscribes to the
// clock's events. The job of DisplayClock is
// to display the current time
public class DisplayClock
{
   // Given a clock, subscribe to
   // its SecondChangeHandler event
   public void Subscribe(Clock theClock)
   {
      theClock.SecondChange +=
         new Clock.SecondChangeHandler(TimeHasChanged);
   }

   // The method that implements the
   // delegated functionality
   public void TimeHasChanged(
      object theClock, TimeInfoEventArgs ti)
   {
      Console.WriteLine("Current Time: {0}:{1}:{2}",
         ti.hour.ToString(),
         ti.minute.ToString(),
         ti.second.ToString());
   }
}
```

http://www.akadia.com/services/dotnet_delegates_and_events.html

LogClock

```
/* ============== Event Subscriber 2 ==================== */

// A second subscriber whose job is to write to a file
public class LogClock
{
   public void Subscribe(Clock theClock)
   {
      theClock.SecondChange +=
         new Clock.SecondChangeHandler(WriteLogEntry);
   }

   // This method should write to a file
   // we write to the console to see the effect
   // this object keeps no state
   public void WriteLogEntry(
      object theClock, TimeInfoEventArgs ti)
   {
      Console.WriteLine("Logging to file: {0}:{1}:{2}",
         ti.hour.ToString(),
         ti.minute.ToString(),
         ti.second.ToString());
   }
}
```

http://www.akadia.com/services/dotnet_delegates_and_events.html

Publisher-Subscriber Template

Publisher

- **Definition of delegate**
  - *One line*
  - „template” for a function that all subscribers will have to implement
  - Usually called „Handler”
  - By convension should return void and it’s parameters should be: object and EventArgs class (or inherited)
  - Example: `public delegate SthHappenedHandler(object o, SpecialEventArgs e);`

- **Event**
  - *One line*
  - Event rised by the publisher
  - General format: public event NameOfDelegate NameOfEvent;
  - Example: `public event SthHappenedHandler SthHappened;`

- **Rising Event Method**
  - Method which fires an event
  - We have to check if someone subscribed to fired event
  - Example: `public void OnSthHappened(SpecialEventArgs e)`
  - `{ if(NameOfEvent!= null) NameOfEvent(this, e); }`

- **Invoke rising-event-method**
  - Remember to call rising event function somewhere!
  - Example:
```
void Publish()
{SpecialEventArgs ev = new SpecialEventArgs(„additional value”);
OnSthHappened(ev); }
```

Contener & Subscriber

[Contener]

- A class that will encapsulate some values that we want to send with an event
- In theory may be omitted; basic types (int, double, string, object…) can be used instead
- But by convention should be of type EventArgs or inherit from EventArgs
- Proposal: use public readonly fields for stored values + create constructor
- Example: `class SpecialEventArgs : EventArgs {`
  `public readonly string add_value;`
  `SpecialEventArgs(string s){add_value=s;} }`

[Subscriber]

- Should implement delegate from Publisher
- Write function with the same parameter / return value as delegate in publisher, that will be called when event will be rised
  - Example: `public DoSomething(object o, SpecialEventArgs e) {`
  - `Console.WriteLine(e.add_value.ToString()); }`

- Remeber to subscribe to the event!
  - You should be able to subscribe or unsubscribe to any event
  - Preferably use += operator
  - Example: `void Subscribe(Publisher publisher){publisher.SthHappened+= new Publisher.SthHappenedHandler(DoSomething);}`

Output parameters

Output parameters

- A parameter declared with an out modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.
- Every output parameter of a method must be definitely assigned before the method returns.
- Output parameters are typically used in methods that produce multiple return values.

```
class OutReturnExample
   {
        static void Method(out int i, out string s1, out string s2)
        {
            i = 44;
            s1 = "I've been returned";
            s2 = null;
        }
        static void Main()
        {
            int value;
            string str1, str2;
            Method(out value, out str1, out str2);
            // value is now 44
            // str1 is now "I've been returned"
            // str2 is (still) null;
        }
   }
```


WYDZIAŁ FIZYKI
POLITECHNIKA WARSZAWSKA
POLITECHNIKA WARSZAWSKA
WARSAW UNIVERSITY OF TECHNOLOGY

File write

File writing

- Similarly to Java we use streams:

```
class FileSaveExample
   {

       static void Main()
       {
           string filename = “file.txt”;
          string string_to_write = “any text”;

          FileStream fileStream;
          StreamWriter streamWriter;

           fileStream = new FileStream(filename, FileMode.Create);
           streamWriter = new StreamWriter(fileStream);

           streamWriter.WriteLine(string_to_write);
            streamWriter.Flush();

           streamWriter.Close();
           fileStream.Close();
       }
   }
```

Task

(see separate file linked on the webpage)

List

PublisherList

Item 1
Item 2
Item 3
…

Each time change happens

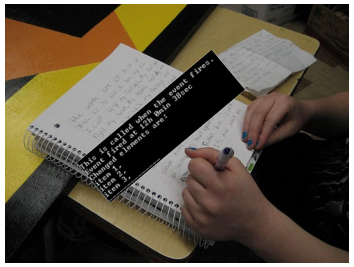
EVENT
list_of_chages, date+time

DisplayChanges

```
This is called when the event fires.
Event fired at 12h 0min 38sec
Changed elements are:
item 1,
item 2,
item 3,
---------------
```

EVENT
list_of_chages, date+time

SaveToFile



Write a list (collection type, it can inherit from ArrayList), which informs about all the changes inside it: it sends information which elements were changed (the list [ArrayList] of changed elements) as well as date and time when the change happened.

References

Akadia

Examples shown today taken from:
http://www.akadia.com/services/dotnet_delegates_and_events.html

Multiple functions with return

When we have multiple functions in one delegate (+=) only **the value of the last function added is returned**!

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

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

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

var b = new BinaryOp(Add);
b += new BinaryOp(Multiply);


var results = b(2,3); // will return 6, the result for multipliciation
```

But, with a little bit of trickery and casting, you can get all of the results like this:

```
var results = b.GetInvocationList().Select(x => (int)x.DynamicInvoke(2, 3));
foreach (var result in results)
    Console.WriteLine(result);
```

THE END

dr inż. Małgorzata Janik

malgorzata.janik@pw.edu.pl