Lecture 3. Abstract programming. Delegates

Programming II

School of Business Informatics
Autumn 2016

(: An optimist says: "The glass is half-full"
A pessimist says: "The glass is half-empty"
A programmer says: "The glass is twice as large as necessary" :)

Canvas registration

- Registration will close this evening
- Make sure your profile full name is adjusted according to the template (Account - Settings - Edit settings)
- Students that cannot be identified by their name will be deleted from the system

- Don't repeat yourself (DRY)
- Separate logic and user interface
- Design small classes. Ideally: each class should have only one responsibility
- Don't make classes strongly coupled to each other

Scenario 1. References to external classes and methods


```
1   public class DataProcessor
2   {
3       public void Calculate()
4       {
5           Console.WriteLine("Started");
6           // Some computation happens here
7           // ...
8           Console.WriteLine("Complete");
9       }
10
11      public void Calculate(System.IO.FileStream fs)
12      {
13          fs.Write(...);
14      }
15  }
```

Scenario 1. References to external classes and methods


```
1   public class DataProcessor
2   {
3       public void Calculate()
4       {
5           Console.WriteLine("Started");
6           // Some computation happens here
7           // ...
8           Console.WriteLine("Complete");
9       }
10
11      public void Calculate(System.IO.FileStream fs)
12      {
13          fs.Write(...);
14      }
15  }
```

Calls to Console and FileStream methods have to be abstracted

Scenario 2. Asynchronous events

The application does not know in advance when these events will occur.

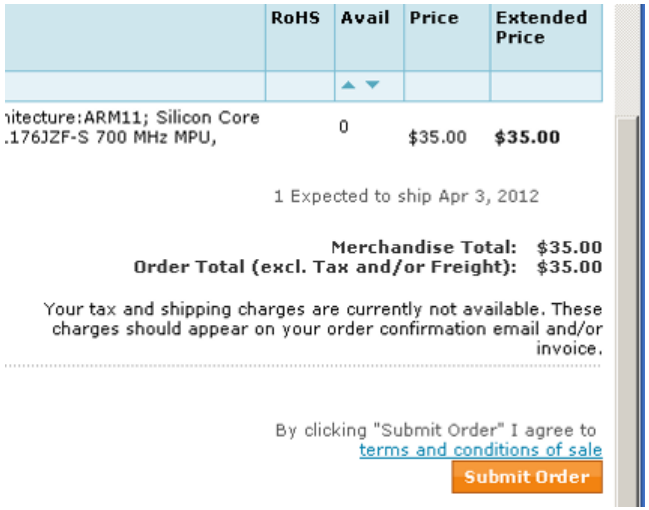
- Timer elapsed
- Mouse button clicked
- Key pressed
- File downloaded
- etc.

Scenario 3. Many handlers for a single event

Online-store:

When a user clicks "Submit order" a number of actions should be executed:

- Save order information to DB
- Send e-mail with confirmation
- Redirect user to another page
- Send order info to the warehouse management system
- Process payment
- and possibly some other

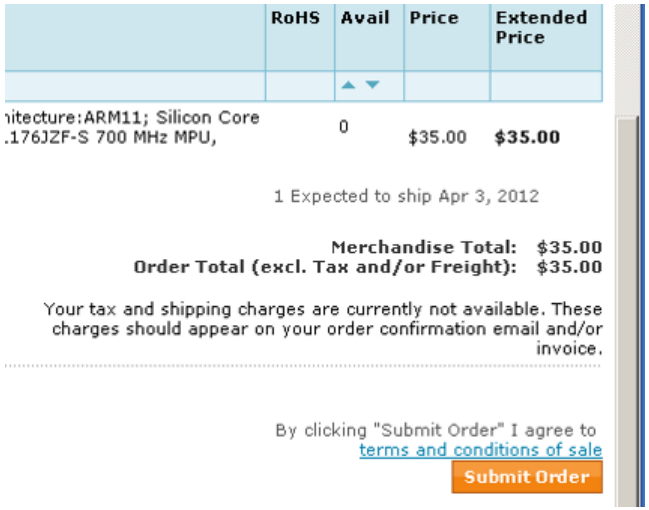


Scenario 3. Many handlers for a single event

Online-store:

When a user clicks "Submit order" a number of actions should be executed:

- Save order information to DB
- Send e-mail with confirmation
- Redirect user to another page
- Send order info to the warehouse management system
- Process payment
- and possibly some other



An efficient way to assign many handlers to a single event is required

Scenario 4. Passing "logic" to functions


```
1
2   var l = new List<int>();
3
4   // Simple case: passing data
5   l.Remove(10);
6
7   // How to make the following calls?
8   l.Remove(<all even elements>);
9   l.Find(<all simple numbers>);
```

Achieving loose coupling

To make OO software easily changeable and extensible it is important to design classes that are independent from each other (loosely coupled classes), for example:

- A class performing mathematical calculations should not be limited to a particular input-output system (file or console)
- A class in a game application that implements game logic should not be strongly coupled with the rendering engine

- Delegates
- Inheritance
- Interfaces

Delegates in C#

- A delegate is a function template (like a class is an object template)
- Delegate declaration:

```
1    public delegate int ProcessDataCallback(
         string data);
```

- Delegate declaration does not allocate memory
- A delegate is a reference type

Delegate instances

- A delegate instance is a variable of the delegate type
- Delegate instances can be declared inside classes, methods or passed as parameters
- The default value of a delegate instance is null

```
1  public ProcessDataCallback DataProcessed;
```

Assigning handlers

- A handler is a concrete method that is linked to a delegate instance at runtime
- Several handlers can be linked to the same delegate instance
- Each handler must match the delegate in return type and parameters

```
1  // Handler function
2  public int OnDataProcessed(string data) { ... }
3
4    // Full syntax
5    processor.DataProcessed += new
         ProcessDataCallback(OnDataProcessed);
6
7    // Short syntax
8    processor.DataProcessed += OnDataProcessed;
```

Making calls through delegate instances

- Calls can be made through delegate instances similar to normal functions
- Before making the call it must be ensured that at least one handler is assigned to a delegate instance:

```
1   void SomeMethod
2   {
3       // ... int result; string data
4
5       // Pre C# 6.0
6       if (DataProcessed != null)
7           result = DataProcessed(data);
8
9       // C# 6.0
10      result = DataProcessed?.Invoke(data);
11  }
```

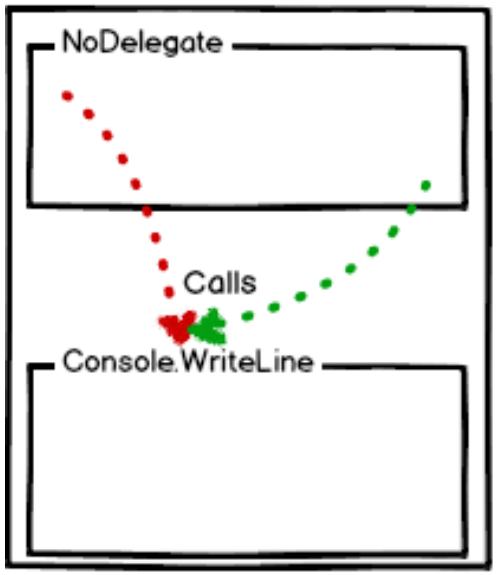
Key idea behind using delegates

A class that makes calls through a delegate instance does not know until runtime which outer methods will be executed

The concept is also known as “late binding”

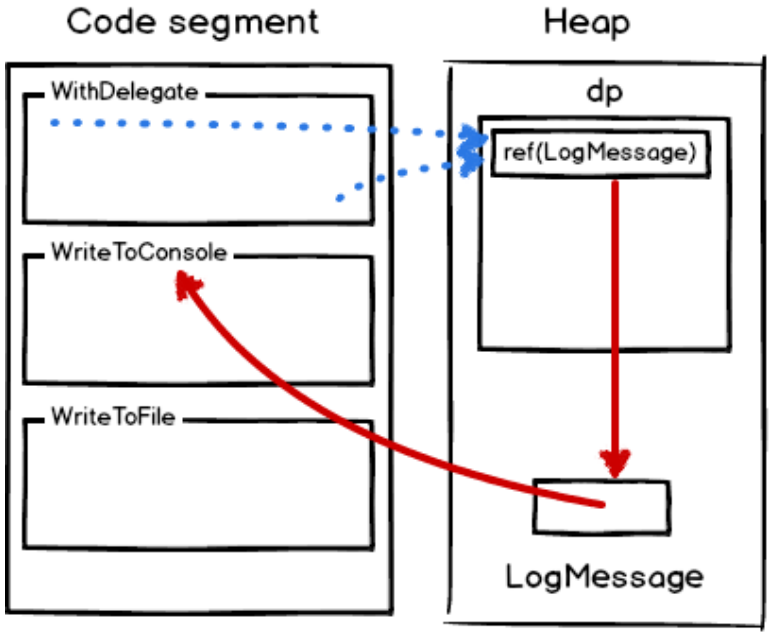
Calling methods without delegates

Code segment



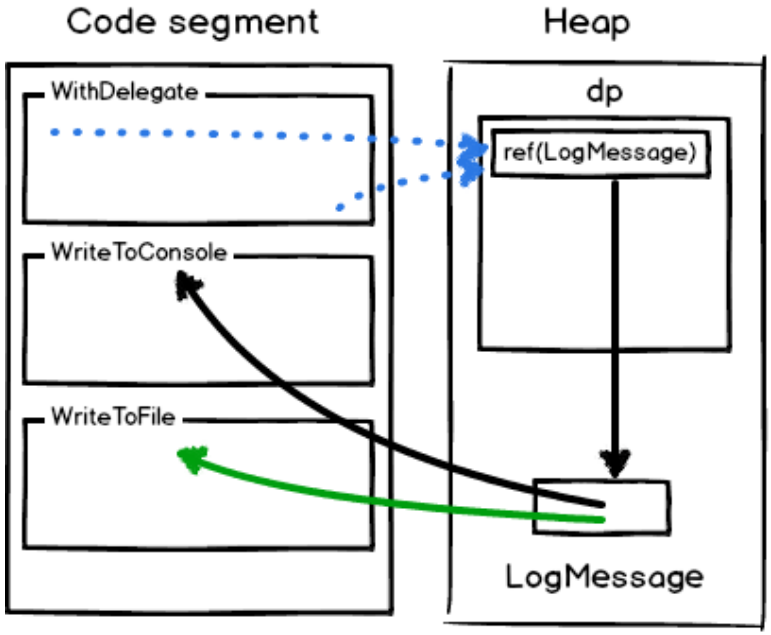

```
public delegate void LogMessageCallback(string message);

public class DataProcessor
{
   public LogMessageCallback LogMessage;

   public void NoDelegate()
   {
      Console.WriteLine("Starting calculation");
      // ... Calculation goes here ...
      Console.WriteLine("Calculation done");
   }

   public void WithDelegate()
   {
      LogMessage("Starting calculation");
      // ... Calculation goes here ...
      LogMessage("Calculation done");
   }
}
```

Using delegates




```
public class DataProcessor
{
    public LogMessageCallback LogMessage;

    public void WithDelegate()
    {
        LogMessage("Starting calculation");
        // ... Calculation goes here ...
        LogMessage("Calculation done");
    }
}

class Program
{
    static StreamWriter sw = new StreamWriter("output.txt", true);

    static void WriteToConsole(string message)
    {
        Console.WriteLine(message);
    }

    static void WriteToFile(string message)
    {
        sw.WriteLine(message); sw.Flush();
    }

    static void Main(string[] args)
    {
        var dp = new DataProcessor();
        dp.LogMessage = new LogMessageCallback(WriteToConsole);
        dp.WithDelegate();
    }
}
```

Multiple handlers




```
public class DataProcessor
{
    public LogMessageCallback LogMessage;

    public void WithDelegate()
    {
        LogMessage("Starting calculation");
        // ... Calculation goes here ...
        LogMessage("Calculation done");
    }
}

class Program
{
    static StreamWriter sw = new StreamWriter("output.txt", true);

    static void WriteToConsole(string message)
    {
        Console.WriteLine(message);
    }

    static void WriteToFile(string message)
    {
        sw.WriteLine(message); sw.Flush();
    }

    static void Main(string[] args)
    {
        var dp = new DataProcessor();
        dp.LogMessage = new LogMessageCallback(WriteToConsole);
        dp.LogMessage += new LogMessageCallback(WriteToFile);
        dp.WithDelegate();
    }
```

C# events

- An event is a special type of a delegate instance
- In its declaration the “event” keyword is used before delegate name:

```
1 class SomeClass
2 {
3     public event EventHandler Event;
4 }
```

Event

An event plays the same role for a delegate instance as a property for a private field

Delegate instances vs events

public ProcessDataHandler processData

or

public event ProcessDataHandler processData

- Delegate instances can be defined inside methods and passed as parameters
- An event can be fired only inside the class where it is declared
- List of handlers for an event cannot be emptied from outside the class. Each handler has to be added or removed separately.

Syntactic sugar

Syntactic sugar is a special syntax within a programming language that makes things easier to read or express.

For example, array[i] instead of array.GetItem(i), extension methods (check online resources)

Syntactic sugar for delegates

1. Generic delegate templates - Action<T>and Func<T,TRet>
2. Lambda expressions

Generic delegate templates

- **Action<T1, T2, ...>** - a template for a void method that accepts parameters of types T1, T2, ...
- **Action** - a template for a void method with no parameters
- **Func<T1, T2, ..., TRet>** - a template for a method that accepts parameters of types T1, T2, ... and returns a value of type TRet

Lambda expressions


```
1  static bool IsEven(int num)
2  {
3    return num % 2 == 0;
4  }
5
6  static void Main(string[] args)
7  {
8    var list = new List<int> {4, 7, 80, 105, 204};
9    // Without a lambda - specify a separate method
10   var evenNumbers = list.FindAll(IsEven);
11   // Using a lambda expression
12   evenNumbers = list.FindAll(num => num % 2 == 0);
13 }
```