HO
GENT

Delegates en Lambdas

Ludwig Stroobant

Table of Contents

1. Doelstellingen

Na het studeren en maken van de oefeningen van dit hoofdstuk ben je in staat om volgende zaken te herkennen, toe te lichten, te definiëren, toe te passen en te implementeren:

- Delegates en Lambda expressies
- Gebruik van `named functions`
- De basis functionele delegates waaronder Function, Predicate, Action …
- Events

2. Inleiding

Een `delegate` is een referentie type naar een methode met een specifieke handtekening. Deze handtekening kan je beschouwen als een abstracte methode declaratie waar, bij declaratie van de delegate, nog geen concrete implementatie aan vast hangt.

> A delegate is a reference type, like the other reference types you've seen, but instead of referring to an object, a delegate refers to a method. This is called encapsulating the method. When you create the delegate, you specify a method signature and return type; you can encapsulate any matching method with that delegate.
>
> — Learning C# 3.0: Master the fundamentals of C# 3.0 by Jesse Liberty, Brian MacDonald

Een referentie variabele van het delegate type kan je laten refereren naar elke methode met een compatibele handtekening en return type als de delegate declaratie. Deze methode kan je dan aanroepen of uitvoeren via de delegate variabele.

Lambda expressies laten toe om in éénzelfde stap zowel de handtekening als de implementatie van een methode te voorzien. Lambda expressies hebben geen formele identiteit tenzij ze aan een delegate variabele worden toegekend.

> Delegates are used to pass methods as arguments to other methods.
>
> — Microsoft Programming Guide

Delegates laten volgende functionaliteit toe:

- De gecreëerde instantie laat toe om functionaliteit te behandelen als een methode argument, functionaliteit wordt behandeld als data die je kan doorgeven.
- Een functie implementeren zonder dat deze toebehoort aan een klasse.
- Een lambda expressie kan aanzien worden als een object (het creëert een object), de bijhorende functionaliteit kan op vraag uitgevoerd worden.

Met Lambda Expressies wordt een stap gezet in de richting van **functioneel programmeren**, een

specifieke vorm van declaratief programmeren:

- Bij declaratief programmeren wordt source code zodanig geschreven dat deze meer aangeeft wat men van de geschreven code verwacht, met weinig of geen nadruk op de eigenlijke implementatie. Je geeft aan ('declareert') wat je wil doen, niet hoe het gedaan wordt.
- Imperatief programmeren is de tegenhanger van declaratief programmeren. Als je bij declaratief programmeren net aangeeft wat je wil bereiken, kan je imperatief programmeren beschouwen als het lijn per lijn instructies coderen om te bereiken wat moet gedaan worden.

- Declaratief programmeren: "Wat wil ik bereiken"
- Imperatief programmeren: "Hoe wil ik het bereiken"

In dit hoofdstuk ligt de focus op delegates, lambda expressies en events. In een later hoofdstuk gaan we deze lambda expressies gebruiken in de context van "Functioneel Programmeren".

3. Delegates

Volgend voorbeeld toont een declaratie van een delegate type:

```
1    public delegate int PerformCalculation(int x, int y);
```

Elke methode van een klasse of een struct, waarvan de handtekening overeenstemt met het delegate type kan toegekend worden aan een variabele van dat delegate type.

Delegates laten toe om @runtime methode aanroepen aan te passen of nieuwe code in bestaande klassen te injecteren.

Delegates maken het mogelijk om een methode mee te geven als een parameter, en zijn zo ideaal om een callback methode te definiëren.

Als voorbeeld zou je een methode kunnen schrijven die twee objecten met elkaar vergelijkt. Deze methode kan op zijn beurt gebruikt worden in een delegate voor een sorteer algoritme. Op deze manier staat het sorteeralgoritme los van de methode die de vergelijk uitvoert en kan dit sorteeralgoritme meer algemeen worden toegepast.

3.1. Declaratie

Een delegate is een referentie type dat een methode inkapselt. Het type van een delegate wordt gedefinieerd door zijn naam.

Volgend voorbeeld declareert een delegate met identifier `Del`. Deze delegate kan een methode inkapselen die een `String` parameter en een `void` return type:

```
1    public delegate void Del(string message);
```

Een delegate van type `Del` kan geïnstantieerd worden door de naam van de methode die de delegate zal inkapelen eraan toe te kennen. Ook een lambda expressie kan toegekend worden aan zo een variabele en instantieert zo een delegate van type `Del`.

Van zodra de delegate geïnstantieerd is kan deze gebruikt worden om de aan de delegate gekoppelde methode uit te voeren. Parameters worden één-op-één doorgegeven en de return value van de methode wordt teruggegeven.

Een delegate kan je aanzien als een *wrapper* van een andere methode.

```
1 // Create a method for a delegate.
2 public static void DelegateMethod(string message)
3 {
4     Console.WriteLine(message);
5 }
```

```
1     // Instantiate the delegate.
2     // Del handler = new Del(DelegateMethod); // C# 1.0 and later
3     Del handler = DelegateMethod; // C# 2.0 and later
4
5     // Call the delegate.
6     handler("Hello World");
```

> When a delegate is used, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.
>
> — Microsoft Programming Guide

Een delegate kan zowel klasse- als instantiemethodes inkapselen. Als een delegate een instantie methode inkapselt, dan refereert de delegate zowel naar de instantie als naar de methode. De delegate heeft echter geen enkele notie van het instantie type, behalve de methode zelf. Als de delegate een static methode inkapselt, refereert ze enkel naar deze methode.

4. Anonieme methodes

Een anoineme methode is een methode waarvan de handtekening, behalve de naam, vastligt. Zo een methode kan gebruikt worden om een delegate te instantiëren.

```
1     public class HelloWorldAnonymousMethod
2     {
3         private delegate void HelloWorld(string someone);
4
```

```
5         public void SayHello()
6         {
7             // Anonymous method
8             HelloWorld frenchGreeting = delegate(string someone) ①
9             {
10            Console.WriteLine("Salut " + someone);
11            };
12
13            frenchGreeting("Fred");
14        }
15
16        public static void Main(string[] args)
17        {
18            HelloWorldAnonymousMethod myApp = new HelloWorldAnonymousMethod();
19            myApp.SayHello();
20        }
21    }
```

5. Lambda Expressies

Een lambda expressie is een expressie die je toelaat om in één stap een anonieme methode te declareren en te instantiëren.

Met een lambda expressie kan je handtekening van een anonieme methode declareren, deze methode implementeren en instantiëren. Een lambda expressie wordt vaak gebruikt om een anonieme methode als argument aan een methode mee te geven.

Lambda expressies laten ons toe functionaliteit door te geven als methode argument, functionaliteit wordt zo behandeld als data die je kan doorgeven.

```
1     public class HelloWorldLambdaExpression
2     {
3         private delegate void HelloWorld(string someone);
4         private const string Prefix = "Hello ";
5
6         private void SayHello(HelloWorld greeting)
7         {
8             greeting("Peter");
9         }
10
11        public static void Main(string[] args)
12        {
13            HelloWorldLambdaExpression myApp = new HelloWorldLambdaExpression();
14
15            HelloWorld dutchGreeting = (string someone) => ①
16            {
17                string name = someone;
```

```
18                 Console.WriteLine(Prefix + name);
19             };
20
21             myApp.SayHello(dutchGreeting);
22
23             myApp.SayHello(s => Console.WriteLine(Prefix + s)); ②
24         }
25     }
```

① De lambda expressie voorziet in de implementatie van de delegate en creëert er in dezelfde stap ook een instantie van. Het resultaat van de lambda expressie wordt hier toegekend aan een variabel van het delegate type.

② Ook de anonieme methode uit deze lambda expressie kan worden meegegeven als parameter van de methode `SayHello`.

5.1. Syntax van Lambda Expressies

Een lambda expressie bevat volgende onderdelen:

```
(parameterlijst) => {statements}
```

- **parameterlijst**: een parameterlijst ingesloten in haakjes

  Een formele parameterlijst is een lijst van parameters zoals je deze hebt bij de declaratie van een methode: het zijn de parameters die de methode verwacht. Bij een lambda expressie zijn dit de parameters die de lambda expressie verwacht.

C# beschikt over 'variable type inference' waardoor de compiler vaak uit de context het type van de parameters kan afleiden. Je kan dit type dus meestal weglaten bij de parameters. Als er maar één parameter is mag je ook de haakjes weglaten.

- Het pijltje is de lambda declaration operator: ⇒
- Een body, die kan bestaan uit één enkele expressie of uit een statement blok. Als je één enkele expressie specifieert zonder statement blok, dan zal de runtime deze expressie evalueren en het resultaat teruggeven.

Onderstaande lambda expressie is equivalent aan voorgaand voorbeeld:

```
1 HelloWorld dutchGreeting = someone -> System.out.println("Hello " + someone);
```

6. Using Named Methods

Lambda expressies kan je aanzien als anonieme methodes.

In sommige gevallen doet een lambda expressie niets meer dan het aanroepen van een bestaande methode. In deze gevallen wordt de code leesbaarder door bij het instantiëren van de delegate rechtstreeks door te mappen naar die methode.

Methode mapping laat je toe om te refereren naar een bestaande methode gebruik makende van zijn naam. Op die manier zorgt methode mapping voor compacte en eenvoudig leesbare code voor methodes die al een naam hebben.

```
1     public class HelloWorldMethodMapping
2     {
3         // Declaration of a delegate HelloWorld
4         private delegate void HelloWorld(string someone);
5         private string Prefix = "Hello ";
6
7         private void SayHello(HelloWorld greeting) // Delegate as parameter of a
  function
8         {
9             greeting("Peter"); // Invoke the delegate
10        }
11
12        private void InstanceMethod(String s)
13        {
14            // Notice this instance method still has access to the instance
  variables
15            Console.WriteLine(Prefix + s);
16        }
17
18        public static void Main(string[] args)
19        {
20            HelloWorldMethodMapping myApp = new HelloWorldMethodMapping();
21
22            // Mapping of static method to delegate
23            myApp.SayHello(Console.WriteLine); ①
24
25            // Mapping of instance method to delegate
26            myApp.SayHello(myApp.InstanceMethod); ②
27        }
28    }
```

① Delegate mappen aan een static methode

② Delegate mappen aan een instance methode

Merk op dat het return type en parameterlijst van de referentie methode exact gelijk moet zijn aan deze van de methode gedeclareerd in de delegate!

Soort	Voorbeeld
Een referentie naar een klasse methode (`static` methode)	ContainingClass.StaticMethodName
Een referentie naar een instantie methode van een specifiek object. De instantie methode van dat object zal aangeroepen worden, het argument wordt als argument doorgegeven.	containingObject::InstanceMethodName

7. Multicast Delegates: Hoe delegates combineren?

Een eigenschap van delegates is dat meerdere objecten kunnen toegekend worden aan één delegate instantie. Hiervoor wordt de `+` operator gebruikt.

Een *multicast delegate* omvat een lijst van toegekende delegates. Wanneer de multicast delegate aangeroepen wordt, wordt de lijst van delegates aangeroepen, in de volgorde waarin ze werden toegekend. Enkel delegates van hetzelfde type kunnen gecombineerd worden.

Zoals de `+` operator kan de `-` operator gebruikt worden om componenten uit een multicast delegate te verwijderen.

```
1     // Define a custom delegate that has a string parameter and returns void.
2     delegate void CustomDel(string s);
3
4     public class MulticastDelegates
5     {
6         // Define two methods that have the same signature as CustomDel.
7         static void Hello(string s)
8         {
9             Console.WriteLine($"  Hello, {s}!");
10        }
11
12        static void Goodbye(string s)
13        {
14            Console.WriteLine($"  Goodbye, {s}!");
15        }
16
17        public static void Main()
18        {
19            // Declare instances of the custom delegate.
20            CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;
21
22            // In this example, you can omit the custom delegate if you
23            // want to and use Action<string> instead.
24            //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;
25
26            // Create the delegate object hiDel that references the
```

```
            // method Hello.
            hiDel = Hello;

            // Create the delegate object byeDel that references the
            // method Goodbye.
            byeDel = Goodbye;

            // The two delegates, hiDel and byeDel, are combined to
            // form multiDel.
            multiDel = hiDel + byeDel;

            // Remove hiDel from the multicast delegate, leaving byeDel,
            // which calls only the method Goodbye.
            multiMinusHiDel = multiDel - hiDel;

            Console.WriteLine("Invoking delegate hiDel:");
            hiDel("A");
            Console.WriteLine("Invoking delegate byeDel:");
            byeDel("B");
            Console.WriteLine("Invoking delegate multiDel:");
            multiDel("C");
            Console.WriteLine("Invoking delegate multiMinusHiDel:");
            multiMinusHiDel("D");
        }
    }
    /* Output:
    Invoking delegate hiDel:
      Hello, A!
    Invoking delegate byeDel:
      Goodbye, B!
    Invoking delegate multiDel:
      Hello, C!
      Goodbye, C!
    Invoking delegate multiMinusHiDel:
      Goodbye, D!
    */
```

8. Basis delegates

C# voorziet ons van enkele basis delegates, wij bekijken de generieke delegates Func, Action en Predicate.

De letter `T` en `TResult` in onderstaand overzicht geven aan dat het gaat om generieke delegates. Deze letters stellen de types voor die later pas ingevuld worden (net zoals bij het gebruik van een List<T>).

Onderstaand overzicht is een fragment dat werd overgenomen uit de Microsoft .NET API.

8.1. Func<T,TResult> Delegate

Encapsulates a method that has one parameter and returns a value of the type specified by the TResult parameter.

```
1 public delegate TResult Func<in T,out TResult>(T arg);
```

8.1.1. Type Parameters

`T`

The type of the parameter of the method that this delegate encapsulates.

`TResult`

The type of the return value of the method that this delegate encapsulates. This type parameter is covariant. That is, you can use either the type you specified or any type that is more derived.

8.1.2. Parameters

arg T

The parameter of the method that this delegate encapsulates.

Return Value

TResult

The return value of the method that this delegate encapsulates.

8.1.3. Examples

The following example demonstrates how to declare and use a Func<T,TResult> delegate. This example declares a Func<T,TResult> variable and assigns it a lambda expression that converts the characters in a string to uppercase. The delegate that encapsulates this method is subsequently passed to the Enumerable.Select method to change the strings in an array of strings to uppercase.

```
 1 // Declare a Func variable and assign a lambda expression to the
 2 // variable. The method takes a string and converts it to uppercase.
 3 Func<string, string> selector = str => str.ToUpper();
 4
 5 // Create an array of strings.
 6 string[] words = { "orange", "apple", "Article", "elephant" };
 7
 8 // Query the array and select strings according to the selector method.
 9 IEnumerable<String> aWords = words.Select(selector);
10
11 // Output the results to the console.
12 foreach (String word in aWords)
```

```
13     Console.WriteLine(word);
```

This code example produces the following output:

```
ORANGE
APPLE
ARTICLE
ELEPHANT
```

8.1.4. Remarks

You can use this delegate to represent a method that can be passed as a parameter without explicitly declaring a custom delegate. The encapsulated method must correspond to the method signature that is defined by this delegate. This means that the encapsulated method must have one parameter that is passed to it by value, and that it must return a value.

To reference a method that has one parameter and returns void, use the generic `Action<T>` delegate instead.

When you use the Func<T,TResult> delegate, you do not have to explicitly define a delegate that encapsulates a method with a single parameter. For example, the following code explicitly declares a delegate named ConvertMethod and assigns a reference to the UppercaseString method to its delegate instance.

```
1  using System;
2
3  delegate string ConvertMethod(string inString);
4
5  public class DelegateExample
6  {
7     public static void Main()
8     {
9        // Instantiate delegate to reference UppercaseString method
10       ConvertMethod convertMeth = UppercaseString;
11       string name = "Dakota";
12       // Use delegate instance to call UppercaseString method
13       Console.WriteLine(convertMeth(name));
14    }
15
16    private static string UppercaseString(string inputString)
17    {
18       return inputString.ToUpper();
19    }
20 }
```

The following example simplifies this code by instantiating the Func<T,TResult> delegate instead of explicitly defining a new delegate and assigning a named method to it.

```
// Instantiate delegate to reference UppercaseString method
Func<string, string> convertMethod = UppercaseString;
string name = "Dakota";
// Use delegate instance to call UppercaseString method
Console.WriteLine(convertMethod(name));

string UppercaseString(string inputString)
{
   return inputString.ToUpper();
}

// This code example produces the following output:
//
//    DAKOTA
```

You can also use the Func<T,TResult> delegate with anonymous methods in C#, as the following example illustrates. (For an introduction to anonymous methods, see Anonymous Methods.)

```
Func<string, string> convert = delegate(string s)
   { return s.ToUpper();};

string name = "Dakota";
Console.WriteLine(convert(name));

// This code example produces the following output:
//
//    DAKOTA
```

You can also assign a lambda expression to a Func<T,TResult> delegate, as the following example illustrates. (For an introduction to lambda expressions, see Lambda Expressions and Lambda Expressions.)

```
Func<string, string> convert = s => s.ToUpper();

string name = "Dakota";
Console.WriteLine(convert(name));

// This code example produces the following output:
//
//    DAKOTA
```

The underlying type of a lambda expression is one of the generic Func delegates. This makes it possible to pass a lambda expression as a parameter without explicitly assigning it to a delegate. In particular, because many methods of types in the System.Linq namespace have Func<T,TResult> parameters, you can pass these methods a lambda expression without explicitly instantiating a Func<T,TResult> delegate.

8.2. Other Func delegates

Similar Func delegates exist with none, one, two or more parameters:

- Func<TResult>
- Func<T,TResult>
- Func<T1,T2,TResult>
- Func<T1,T2,T3,TResult>
- Func<T1,T2,T3,T4,TResult>

8.3. Action<T> Delegate

Encapsulates a method that has a single parameter and does not return a value.

```
1 public delegate void Action<in T>(T obj);
```

8.3.1. Type Parameters

`T`

The type of the parameter of the method that this delegate encapsulates.

8.3.2. Parameters

obj T

The parameter of the method that this delegate encapsulates.

8.3.3. Examples

The following example demonstrates the use of the Action<T> delegate to print the contents of a List<T> object. In this example, the Print method is used to display the contents of the list to the console. In addition, the C# example also demonstrates the use of anonymous methods to display the contents to the console.

Note that the example does not explicitly declare an Action<T> variable. Instead, it passes a reference to a method that takes a single parameter and that does not return a value to the List<T>.ForEach method, whose single parameter is an Action<T> delegate.

Similarly, in the C# example, an Action<T> delegate is not explicitly instantiated because the signature of the anonymous method matches the signature of the Action<T> delegate that is expected by the List<T>.ForEach method.

```
1     List<string> names = new List<string>();
2     names.Add("Bruce");
3     names.Add("Alfred");
```

```
    names.Add("Tim");
    names.Add("Richard");

    // Display the contents of the list using the Print method.
    names.ForEach(Print);

    // The following demonstrates the anonymous method feature of C#
    // to display the contents of the list to the console.
    names.ForEach(delegate(string name)
        {
            Console.WriteLine(name);
        });

    void Print(string s)
    {
        Console.WriteLine(s);
    }

    /* This code will produce output similar to the following:
    * Bruce
    * Alfred
    * Tim
    * Richard
    * Bruce
    * Alfred
    * Tim
    * Richard
    */
```

8.3.4. Remarks

You can use the Action<T> delegate to pass a method as a parameter without explicitly declaring a custom delegate. The encapsulated method must correspond to the method signature that is defined by this delegate. This means that the encapsulated method must have one parameter that is passed to it by value, and it must not return a value. (In C#, the method must return void. It can also be a method that returns a value that is ignored.)

Typically, such a method is used to perform an operation.

To reference a method that has one parameter and returns a value, use the generic Func<T,TResult> delegate instead.

When you use the Action<T> delegate, you do not have to explicitly define a delegate that encapsulates a method with a single parameter. For example, the following code explicitly declares a delegate named DisplayMessage and assigns a reference to either the WriteLine method or the ShowWindowsMessage method to its delegate instance.

```
using System;
using System.Windows.Forms;
```

```
3
4 delegate void DisplayMessage(string message);
5
6 public class TestCustomDelegate
7 {
8    public static void Main()
9    {
10       DisplayMessage messageTarget;
11
12       if (Environment.GetCommandLineArgs().Length > 1)
13          messageTarget = ShowWindowsMessage;
14       else
15          messageTarget = Console.WriteLine;
16
17       messageTarget("Hello, World!");
18    }
19
20    private static void ShowWindowsMessage(string message)
21    {
22       MessageBox.Show(message);
23    }
24 }
```

The following example simplifies this code by instantiating the Action<T> delegate instead of explicitly defining a new delegate and assigning a named method to it.

```
1 using System;
2 using System.Windows.Forms;
3
4 public class TestAction1
5 {
6    public static void Main()
7    {
8       Action<string> messageTarget;
9
10       if (Environment.GetCommandLineArgs().Length > 1)
11          messageTarget = ShowWindowsMessage;
12       else
13          messageTarget = Console.WriteLine;
14
15       messageTarget("Hello, World!");
16    }
17
18    private static void ShowWindowsMessage(string message)
19    {
20       MessageBox.Show(message);
21    }
22 }
```

You can also use the Action<T> delegate with anonymous methods in C#, as the following example illustrates. (For an introduction to anonymous methods, see Anonymous Methods.)

```
using System;
using System.Windows.Forms;

public class TestAnonMethod
{
   public static void Main()
   {
      Action<string> messageTarget;

      if (Environment.GetCommandLineArgs().Length > 1)
         messageTarget = delegate(string s) { ShowWindowsMessage(s); };
      else
         messageTarget = delegate(string s) { Console.WriteLine(s); };

      messageTarget("Hello, World!");
   }

   private static void ShowWindowsMessage(string message)
   {
      MessageBox.Show(message);
   }
}
```

You can also assign a lambda expression to an Action<T> delegate instance, as the following example illustrates. (For an introduction to lambda expressions, see Lambda Expressions.)

```
using System;
using System.Windows.Forms;

public class TestLambdaExpression
{
   public static void Main()
   {
      Action<string> messageTarget;

      if (Environment.GetCommandLineArgs().Length > 1)
         messageTarget = s => ShowWindowsMessage(s);
      else
         messageTarget = s => Console.WriteLine(s);

      messageTarget("Hello, World!");
   }

   private static void ShowWindowsMessage(string message)
   {
      MessageBox.Show(message);
```

```
21     }
22 }
```

The ForEach and ForEach methods each take an Action<T> delegate as a parameter. The method encapsulated by the delegate allows you to perform an action on each element in the array or list.

8.4. Other Action delegates

Similar Action delegates exist with none, one, two or more parameters:

- Action
- Action<T>
- Action<T1,T2>
- Action<T1,T2,T3>
- Action<T1,T2,T3,T4>
- Action<T1,T2,T3,T4,T5>

8.5. Predicate<T> Delegate

Represents the method that defines a set of criteria and determines whether the specified object meets those criteria.

```
1 public delegate bool Predicate<in T>(T obj);
```

8.5.1. Type Parameters

T

The type of the object to compare.

8.5.2. Parameters

obj T

The object to compare against the criteria defined within the method represented by this delegate.

Return Value

Boolean

true if obj meets the criteria defined within the method represented by this delegate; otherwise, false.

8.5.3. Examples

The following code example uses a Predicate<T> delegate with the Array.Find method to search an

array of Point structures. The example explicitly defines a Predicate<T> delegate named predicate and assigns it a method named FindPoints that returns true if the product of the Point.X and Point.Y fields is greater than 100,000.

It is customary to use a lambda expression rather than to explicitly define a delegate of type Predicate<T>, as the second example illustrates.

```
using System;
using System.Drawing;

public class Example
{
   public static void Main()
   {
      // Create an array of Point structures.
      Point[] points = { new Point(100, 200),
                         new Point(150, 250), new Point(250, 375),
                         new Point(275, 395), new Point(295, 450) };

      // Define the Predicate<T> delegate.
      Predicate<Point> predicate = FindPoints;

      // Find the first Point structure for which X times Y
      // is greater than 100000.
      Point first = Array.Find(points, predicate);

      // Display the first structure found.
      Console.WriteLine("Found: X = {0}, Y = {1}", first.X, first.Y);
   }

   private static bool FindPoints(Point obj)
   {
      return obj.X * obj.Y > 100000;
   }
}
// The example displays the following output:
//        Found: X = 275, Y = 395
```

The following example is identical to the previous example, except that it uses a lambda expression to represent the Predicate<T> delegate. Each element of the points array is passed to the lambda expression until the expression finds an element that meets the search criteria. In this case, the lambda expression returns true if the product of the X and Y fields is greater than 100,000.

```
using System;
using System.Drawing;

public class Example
{
```

```
   public static void Main()
   {
      // Create an array of Point structures.
      Point[] points = { new Point(100, 200),
                         new Point(150, 250), new Point(250, 375),
                         new Point(275, 395), new Point(295, 450) };

      // Find the first Point structure for which X times Y
      // is greater than 100000.
      Point first = Array.Find(points, x => x.X * x.Y > 100000 );

      // Display the first structure found.
      Console.WriteLine("Found: X = {0}, Y = {1}", first.X, first.Y);
   }
}
// The example displays the following output:
//        Found: X = 275, Y = 395
```

8.5.4. Remarks

This delegate is used by several methods of the Array and List<T> classes to search for elements in the collection.

Typically, the Predicate<T> delegate is represented by a lambda expression.

Because locally scoped variables are available to the lambda expression, it is easy to test for a condition that is not precisely known at compile time. This is simulated in the following example, which defines a HockeyTeam class that contains information about a National Hockey League team and the year in which it was founded. The example defines an array of integer values that represent years, and randomly assigns one element of the array to foundedBeforeYear, which is a variable that is locally scoped to the example's Main method. Because locally scoped variables are available to a lambda expression, the lambda expression passed to the List<T>.FindAll method is able to return a HockeyTeam object for each team founded on or before that year.

```
using System;
using System.Collections.Generic;

public class HockeyTeam
{
   private string _name;
   private int _founded;

   public HockeyTeam(string name, int year)
   {
      _name = name;
      _founded = year;
   }

   public string Name {
```

```
16        get { return _name; }
17     }
18
19     public int Founded {
20        get { return _founded; }
21     }
22  }
23
24  public class Example
25  {
26     public static void Main()
27     {
28        Random rnd = new Random();
29        List<HockeyTeam> teams = new List<HockeyTeam>();
30        teams.AddRange( new HockeyTeam[] { new HockeyTeam("Detroit Red Wings", 1926),
31                                           new HockeyTeam("Chicago Blackhawks",
    1926),
32                                           new HockeyTeam("San Jose Sharks", 1991),
33                                           new HockeyTeam("Montreal Canadiens",
    1909),
34                                           new HockeyTeam("St. Louis Blues", 1967) }
    );
35        int[] years = { 1920, 1930, 1980, 2000 };
36
37        int foundedBeforeYear = years[rnd.Next(0, years.Length)];
38        Console.WriteLine("Teams founded before {0}:", foundedBeforeYear);
39
40        foreach (var team in teams.FindAll( x => x.Founded <= foundedBeforeYear))
41           Console.WriteLine("{0}: {1}", team.Name, team.Founded);
42     }
43  }
44  // The example displays output similar to the following:
45  //       Teams founded before 1930:
46  //       Detroit Red Wings: 1926
47  //       Chicago Blackhawks: 1926
48  //       Montreal Canadiens: 1909
```

9. Events

Grafische User Interfaces (GUIs), zoals bv. Microsoft Windows of een web browser, verwachten dat er events kunnen afgehandeld worden. Voorbeelden van zo een events zijn het aanklikken van een knop (button), een selectie maken in een menu, het aflopen van een kopieer operatie ... Er gebeurt iets en daar wil je vanuit je programma op reageren.

De volgorde waarin events optreden zijn niet voorspelbaar. Het systeem lijkt in rust totdat er een event optreedt, en komt tot leven om op een event te reageren.

Vertalen we dit naar software, dan zijn er klassen die willen reageren op events.

Er is één klasse die een event triggert, andere klassen kunnen geïnteresseerd zijn in dit event om erop te reageren. Hoe ze reageren interesseert de klasse die het even triggert niet. Een Button klasse kan zeggen, "Hey, iemand heeft me aangeklikt!". Eén of meerdere andere klassen kunnen hierop reageren.

9.1. Publishing en Subscribing

```
In C#, any object can *publish a set of events* to which other classes can
*subscribe*. When the publishing class *raises an event*, all the subscribed classes
are *notified*. With this mechanism, your object can say, "Here are things I can
notify you about," and other classes might sign up, saying, "Yes, let me know when
that happens."

For example, a button might notify any number of interested observers when it is
clicked.

The button is called the publisher because the button publishes the Click event and
the other classes are the subscribers because they subscribe to the Click event.
```

De klasse die het event publiceert heeft geen enkele info en is ook niet geïnteresseerd in wie dit event afhandelt: de klasse zal enkel het event triggeren. Wie en hoe erop gereageerd wordt is voor deze klasse niet van belang.

Dit ontwerp implementeert het *Observer pattern*.

9.2. Clock voorbeeld

Een klasse `Clock` kan geïnteresseerde klassen notificeren als de tijd veranderd (bv per seconde). In eerste instantie zou je geneigd kunnen zijn om diezelfde klasse `Clock` ook de user interface representatie van de tijd te laten afhandelen. Waarom zou je dan werken met delegates om andere klassen op de hoogte te brengen van zo een event?

Het voordeel van het *Observer patroon*, dat toelaat om een event te publiceren en geïnteresseerden te laten inschrijven, is als volgt: de klasse `Clock` is verantwoordelijk om de tijd bij te houden. Moet deze klasse dan ook weten hoe deze informatie (= de tijd) zal gebruikt worden?

Door gebruik te maken van een event (= een seconde die wijzigt) kan het bijhouden van de tijd *ontkoppeld* worden van hoe die informatie gebruikt zal worden (bv. het weergeven van de tijd, het loggen van de tijd in een bestand, een alarm dat moet afgaan, een oven die moet uitschakelen op een bepaald tijdstip ...)

Eender aantal klassen kan interesse tonen in het reageren op het event (= een seconde die wijzigt) en willen hiervan een bericht krijgen (= notify). Om dit bericht te ontvangen kunnen ze zich inschrijven op het event.

Klassen die zich inschrijven op het event van de klasse `Clock` zijn verantwoordelijk om op dit event te reageren. Zij hoeven niet te weten hoe de tijd zelf wordt bijgehouden, dat is de verantwoordelijkheid van de klasse `Clock`. De klasse `Clock` hoeft op zijn beurt niets af te weten van de klassen die reageren op het event, ze stuurt hen enkel een bericht indien het event optrad. Ook de verschillende klassen die reageren op het event kennen elkaar niet. Ze zijn dus ontkoppeld van elkaar.

Door het gebruik van een delegate kan de *publisher* van een *event* worden losgekoppeld van een *subscriber*. Dit is gewenst en maakt de code meer flexibel en robuuster. Het afhandelen van hoe de klok werkt zit ingekapseld in de klasse `Clock`. Het afhandelen van de tijd zit ingekapseld in de klassen die op het event reageren.

Publishers en Subscribers werken dus onafhankelijk van elkaar, wat de mainainability van de code verhoogt.

Design Principle: Strive for loosely coupled designs between objects that interact.

9.3. Events en Delegates

Events worden in C# geïmplementeerd op basis van delegates.

De klasse die het event publiceert zal een delegate definiëren. De klassen die zich inschrijven voor het event moeten:

- een methode implementeren die voldoet aan de delegate handtekening
- een delegate instantiëren die de vorige methode inkapselt.

Als het event optreedt zal de methode van de klasse die zich inschreef op het event uitgevoerd worden via de delegate.

Een methode die een event afhandelt noemt men een *event handler*.

Event handlers in het .NET Framework hebben altijd return type `void` en twee parameters. De eerste parameter is de "source" van het event (typisch is dit het publicerende object). De tweede parameter is een instantie van de klasse `EventArgs` of een subklasse hiervan.

De klasse `EventArgs` is een helper klasse die gebruikt kan worden om extra informatie omtrent het event mee te geven aan de subscribers.

9.4. Solving Delegate Problems with Events

Onderzoek de werking van de voorbeeldcode: `DelegateToEventExample_Clock`.

In dit voorbeeld werd het afhandelen van een gebeurtenis geïmplementeerd op basis van een delegate.

Het event kan afgehandeld worden via een delegate, maar er is een mogelijk probleem met deze code!

Er wordt gebruik gemaakt van een multicast delegate om de verschillende subscribers bij te houden. Stel nu dat de software ontwikkelaar van de code een bug schreef in de klasse `LogCurrentTime`, en per ongeluk de operator (=) gebruikte om te registreren bij het event i.p.v. de operator (+=)?

```
public void Subscribe(Clock theClock)
{
  theClock.SecondChanged =
    new Clock.SecondChangeHandler(WriteLogEntry);
}
```

Deze bug zorgt ervoor dat alle voorgaande subscribers gewist worden. Een klein foutje met grote gevolgen!

Een tweede probleem is dat de delegate `SecondChangeHandler` ook door anderen kan aangeroepen worden.

```
    Console.WriteLine("Calling the method directly!");
    System.DateTime dt = System.DateTime.Now.AddHours(2);
    TimeInfoEventArgs timeInformation =
        new TimeInfoEventArgs(  dt.Hour,dt.Minute,dt.Second);
    theClock.SecondChanged(theClock, timeInformation);
```

Hier werd de delegate `SecondChanged` vanuit de Main methode aangeroepen. De code werkt, maar dit was niet de intentie van diegene die de klasse `Clock` implementeerde.

9.5. Het `event` Keyword

Bovenstaande problemen worden opgelost door het keyword `event`. Dit keyword zorgt ervoor dat de delegate enkel kan aangeroepen worden vanuit de klasse waarin hij gedeclareerd werd. Subscribers kunnen zich vanaf nu enkel registreren of uitschrijven met de operators `+=` en `-=`.

```
public event SecondChangeHandler SecondChanged;
```

9.6. Lambda Expressie

In de code zou je ook gebruik kunnen maken van een lambda expressie:

```
theClock.OnSecondChange +=
    (aClock, ti) =>
    {
        Console.WriteLine("Current Time: {0}:{1}:{2}",
```

```
5                           ti.hour.ToString( ),
6                           ti.minute.ToString( ),
7                           ti.second.ToString( ));
8       };
```