21. Overloaded Operators

In this section we will see how we can use the operators of the C# language on instances of our own classes, or on values of our own structs.

21.1. Why operator overloading?

Lecture 6 - slide 2

In this section we will describe how to program operations that can be called in expressions that make use of the conventional operators (such as **+**, **&**, **>>**, and **!**) of C#. Thus, in a client of a class C, we will provide for notation such as `aC1 + aC2 * aC3` instead of `aC1.Plus(aC2.Mult(aC3))` or (with use of class methods) `C.Plus(aC1, C.Mult(aC2,aC3))`.

Use of operators provides for substantial notational convenience in certain classes

When operator notation is natural for objects or values of type C, clients of C can often be programmed with a more dense and readable notation. The example in Program 21.1 (only on web) provides additional motivation. The class `OperatorsOrNot`, (see Program 21.1 on the web) together with a class `MyInt` (see Program 21.2 on the web) motivate the use of operators in the context of a complete class.

21.2. Overloadable operators in C#

Lecture 6 - slide 3

We have already once studied the operator table of C#, see Section 6.7. In Table 21.1 below we show a version of the operator table of C# (with operator priority and associativity) in which we have emphasized all the operators that can be overloaded in C#.

Level	Category	Operators	Associativity
14	*Primary*	`x.y  f(x)  a[x]  x++  x--` `new  typeof  checked  unchecked  default  delegate`	*left to right*
13	*Unary*	`+  -  !  ~  ++x  --x  (T)x  true  false  sizeof`	*left to right*
12	*Multiplicative*	`*  /  %`	*left to right*
11	*Additive*	`+  -`	*left to right*
10	*Shift*	`<<  >>`	*left to right*
9	*Relational and Type testing*	`<  <=  >  >=  is  as`	*left to right*
8	*Equality*	`==  !=`	*left to right*
7	*Logical/bitwise And*	`&`	*left to right*
6	*Logical/bitwise Xor*	`^`	*left to right*
5	*Logical/bitwise Or*	`|`	*left to right*
4	*Conditional And*	`&&`	*left to right*
3	*Conditional Or*	`||`	*left to right*
2	*Conditional*	`?:`	*right to left*
1	*Assignment*	`=  *=  /=  %=  +=  -=` `<<=  >>=  &=  ^=  |=  ??  =>`	*right to left*

Table 21.1 *The operator priority table of C#. The operators that can be overloaded directly are emphasized.*

All the gray operators in Table 21.1 cannot be overloaded directly. Many of them can, however, be overloaded indirectly, or defined by other means. We will now discuss how this can be done.

A notation similar to `a[x]` (array indexing) can be obtained by use of indexers, see Chapter 19.

The conditional (short circuited) operators `&&` and `||` can be overloaded indirectly by overloading the operators `&` and `|`. In addition, the operators called `true` and `false` must also be provided. `true(x)` tells if `x` counts as boolean true. `false(x)` tells if `x` counts as boolean false. (Notice that `x` belongs to the type - class or struct - in which we have defined the operators). The operators `&&` and `||` are defined by the following equivalences:

- `x && y` is equivalent to `false(x) ? x : (x & y)`
- `x || y` is equivalent to `true(x) ? x : (x | y)`

Thus, when `x && y` is encountered we first evaluate the expression `false(x)`. If the value is *true*, `x` is returned. If it is *false*, `y` is also evaluated, and the value of `x && y` becomes `x & y`. A similar explanation applies for `x || y`.

You can define the unary `true` and `false` operators in your own classes, and hereby control if the object is considered to be *true* or *false* in some boolean contexts. If you define one of them, you will also have to define the other. Recall that an expression of the form `a ? b : c` uses the conditional operator `?:` with the meaning `if a then b else c`.

All the assignment operators, apart from the basic assignment operator `=`, are implicitly overloadable. As an example, the assignment operator `*=` is implicitly overloaded when we explicitly overload the multiplication operator `*`.

The type cast operator `(T)x` can in reality also be overloaded. In a given class C you can define explicit and/or implicit conversion operators that converts to and from C. We will see an example of an explicit type conversion in Program 21.3.

21.3. An example of overloaded operators: Interval

Lecture 6 - slide 4

We will now study the type `Interval`. This type allows us to represent and operate on intervals of integers. The `Interval` type makes a good case for illustration of overloaded operators. We program all interval operations in a functional style. We want intervals to be non-mutable, and the type is therefore programmed as a struct.

An interval is characterized by two integer end points *from* and *to*. The interval [*from* - *to*] denotes the interval that starts in *from* and goes to *to*. The notation [*from* - *to*] is an informal notation which we use to explain the the idea of intervals. In a C# program, the interval [*from* - *to*] is denoted by the expression `new Interval(` *from*`,`*to*`)`. Notice that *from* is not necessarily less than *to*. The following are concrete examples: `[1 - 5]` represents the sequence 1, 2, 3, 4, and 5. `[5 - 1]` represents the sequence 5, 4, 3, 2, and 1. These two sequences are different.

In Program 21.3 we see the struct `Interval`. The private instance variables `from` and `to` represent the interval in a simple and straightforward way, and the constructor is also simple. Just after the constructor there are two properties, `From` and `To`, that access the end points of the interval. In this version of the type it is not possible to construct an empty interval. We have already dealt with this weakness in Section 16.4 (see Program 16.7 versus Program 16.8) in the context of factory methods.

After the two properties we have highlighted a number of overloaded operators. These are our main interest in this section. Notice the syntax for definition of the operators. There are two definitions of the + operator. One of the form `anInterval + i` and one of the form `i + anInterval`. Both have the same meaning, namely addition of `i` to both end-points. Thus `[1 - 5] + 3` and `3 + [1 - 5]` are both equal to the interval `[4 - 8]`.

In similar ways we define multiplication of intervals and integers. We also define subtraction of an integer from an interval (but not the other way around). The shift operators << and >> provide nice notations for moving one of the end-points of an interval. Thus, `[1 - 5] >> 3` is equal to the interval `[1 - 8]`.

Finally, the unary prefix operator `!` reverses an interval (internally, by making an interval with swapped end-points). Thus, `![1 - 5]` is equal to the interval `[5 - 1]`.

The private class `IntervalEnumerator` (shown only in the web version) and the method `GetEnumerator` make it possible to traverse an interval in a convenient way with use of **`foreach`**. Interval traversal is what

makes intervals useful. This is illustrated in Program 21.4. We will, in great details, discuss `IntervalEnumerator` later in this material, see Section 31.6 - in particular Program 31.9.

```
using System;
using System.Collections;

public struct Interval{

  private readonly int from, to;

  public Interval(int from, int to){
    this.from = from;
    this.to = to;
  }

  public int From{
    get {return from;}
  }

  public int To{
    get {return to;}
  }

  public int Length{
    get {return Math.Abs(to - from) + 1;}
  }

  public static Interval operator +(Interval i, int j){
    return new Interval(i.From + j, i.To + j);
  }

  public static Interval operator +(int j, Interval i){
    return new Interval(i.From + j, i.To + j);
  }

  public static Interval operator >>(Interval i, int j){
    return new Interval(i.From, i.To + j);
  }

  public static Interval operator <<(Interval i, int j){
    return new Interval(i.From + j, i.To);
  }

  public static Interval operator *(Interval i, int j){
    return new Interval(i.From * j, i.To * j);
  }

  public static Interval operator *(int j, Interval i){
    return new Interval(i.From * j, i.To * j);
  }

  public static Interval operator -(Interval i, int j){
    return new Interval(i.From - j, i.To - j);
  }

  public static Interval operator !(Interval i){
    return new Interval(i.To, i.From);
  }

  public static explicit operator int[] (Interval i){
    int[] res = new int[i.Length];
    for (int j = 0; j < i.Length; j++) res[j] = i[j];
    return res;
  }
```

```
62
63    private class IntervalEnumerator: IEnumerator{
64      // Details not shown in this version
65    }
66
67    public IEnumerator GetEnumerator (){
68      return new IntervalEnumerator(this);
69    }
70
71  }
```

Program 21.3 *The struct Interval.*

Take a look at Program 21.4 in which we use intervals. Based on the constructed intervals `iv1` and `iv2` we write expressions that involve intervals. These are all highlighted in Program 21.4. Let me explain the expression `!(3 + !iv2 * 2)`. When we evaluate this expression we adhere to normal precedence rules and normal association rules of the operators. We cannot change these rules. Therefore, we first evaluate `!iv2`, which is `[5 - 2]`. Next we evaluate `!iv2 * 2`, which is `[10 - 4]`. To this interval we add 3. This gives the interval `[13 - 7]`. Finally we reverse this interval. The final value is `[7 - 13]`.

Also emphasized in Program 21.3 we show `iv3[0]` and `iv3[iv3.Length-1]`. These expressions use interval indexers. In Exercise 6.1 it is an exercise to program this indexer.Emphasized with **blue** in Program 21.3 and Program 21.4 we show how to program and use an explicit type cast from `Interval` to `int[]`.You should follow the evaluations of all highlighted expressions in Program 21.4 and compare your results with the program output in Listing 21.5.

```
1  using System;
2
3  public class app {
4
5    public static void Main(){
6
7      Interval iv1 = new Interval(17,14),
8               iv2 = new Interval(2,5),
9               iv3;
10
11     foreach(int k in !(3 + iv1 - 2)){
12       Console.Write("{0,4}", k);
13     }
14     Console.WriteLine();
15
16     foreach(int k in !(3 + !iv2 * 2)){
17       Console.Write("{0,4}", k);
18     }
19     Console.WriteLine();
20
21     iv3 = !(3 + !iv2 * 3) >> 2 ;
22     Console.WriteLine("First and last in iv3: {0}, {1}",
23                       iv3[0], iv3[iv3.Length-1]);
24
25     int[] arr = (int[])iv3;
26     foreach(int j in arr){
27       Console.Write("{0,4}", j);
28     }
29
30   }
31
32 }
```

Program 21.4 *A client program of struct Interval.*

```
  15  16  17  18
   7   8   9  10  11  12  13
First and last in iv3: 9, 20
   9  10  11  12  13  14  15  16  17  18  19  20
```

Listing 21.5 *Output from the interval application.*

In Program 21.6 we show yet another example of programming overloaded operators. We overload ==, !=, <, and >. This example brings us back to the playing card class which we have discussed already in Program 12.7 of Section 12.6 and Program 14.3 of Section 14.3.

Emphasized with colors in Program 21.6 we show operators that compare two cards. Notice, as above, that the operator definitions always are static. Also notice that if we define == we also have to define `!=` . The == operator is defined via the `Equals` methods, which is redefined in class `Card` such that it provides value comparison of `Card` instances. If we redefine `Equals` we must also redefine `GetHashCode`. All together, a lot of work! Similarly, if we define <= we have also have to define >= .

Please notice that our redefinition of `Equals` in Program 21.6 is too simple for a real-life program. In Section 28.16 we will see the general pattern for redefinition of the `Equals` instance method.

```
using System;

public enum CardSuite { Spades, Hearts, Clubs, Diamonds };
public enum CardValue { Ace = 1, Two = 2, Three = 3, Four = 4, Five = 5,
                        Six = 6, Seven = 7, Eight = 8, Nine = 9, Ten = 10,
                        Jack = 11, Queen = 12, King = 13};


public class Card{

   private CardSuite suite;
   private CardValue value;

   // Some methods are not shown in this version

   public override bool Equals(Object other){
      return (this.suite == ((Card)other).suite) &&
             (this.value == ((Card)other).value);
   }

   public override int GetHashCode(){
      return (int)suite ^ (int)value;
   }

   public static bool operator ==(Card c1, Card c2){
     return c1.Equals(c2);
   }

   public static bool operator !=(Card c1, Card c2){
     return !(c1.Equals(c2));
   }

   public static bool operator <(Card c1, Card c2){
     bool res;
     if (c1.suite < c2.suite)
        res = true;
     else if (c1.suite == c2.suite)
        res = (c1.value < c2.value);
     else res = false;
        return res;
```

```
41    }
42
43    public static bool operator >(Card c1, Card c2){
44      return !(c1 < c2) && !(c1 == c2);
45    }
46 }
```

Program 21.6 *The class PlayingCard with relational operators.*

The details left out in line 14 of Program 21.6 can be seen in the web-version of the paper.

Exercise 6.1. *Interval indexer*

It is recommended that you use the web edition of the material when you solve this exercise. The web edition has direct links to the class source files, which you should use as the starting point.

The `Interval` type represents an oriented interval [`from` - `to`] of integers. We use the `Interval` example to illustrate the overloading of operators. If you have not already done so, read about the idea behind the struct `Interval` in the course teaching material.

In the client of struct `Interval` we use an indexer to access elements of the interval. For some interval `i`, the expression `i[0]` should access the `from`-value of `i`, and `i[i.Length-1]` should access the `to`-value of `i`.

Where, precisely, is the indexer used in the given client class?

Add the indexer to the struct `Interval` (getter only) which accesses element number *j* (0 <= *j* <= `i.Length`) of an interval `i`.

Hint: Be careful to take the orientation of the interval into account.

Does it make sense to program a setter of this indexer?

Exercise 6.2. *An interval overlap operation*

It is recommended that you use the web edition of the material when you solve this exercise. The web edition has direct links to the class source files, which you should use as the starting point.

In this exercise we continue our work on struct `Interval`, which we have used to illustrate overloaded operators in C#.

Add an `Interal` operation that finds the overlap between two intervals. Your starting point should be the struct Interval. In the version of struct `Interval`, provided as starting point for this exercise, intervals may be empty.

Please analyze the possible overlappings between two intervals. There are several cases that need consideration. The fact that `Interval` is oriented may turn out to be a complicating factor in the solution. Feel free to ignore the orientation of intervals in your solution to this exercise.

Which kind of operation will you chose for the overlapping operation in C# (method, property, indexer,

operator)?

Before you program the operation in C# you should design the *signature* of the operation.

Program the operation in C#, and test your solution in an `Interval` client program. You may chose to revise the Interval client program from the teaching material.

21.4. Some details of operator overloading

Lecture 6 - slide 5

Below we summarize the syntax of operator definition, which overloads a predefined operators symbol.

```
public static return-type operator symbol(formal-par-list){
  body-of-operator
}
```

Syntax 21.1 *The C# syntax for definition of an overloaded operator.*

There are many detailed rules that must be observed when we overload the predefined operator symbols. Some of them were mentioned in Section 21.3. Others are brought up below.

- Operators must be public and static
- One or two formal parameters must occur, corresponding to unary and binary operators
- At least one of the parameters must be of the type to which the operator belongs
- Only value parameters apply
- Some operators must be defined in pairs (either none or both):
  - `==` and `!=` `<` and `>` `<=` and `>=`
- The special unary boolean operators **`true`** and **`false`** define when an object is playing the role as *true* or *false* in relation to the conditional logical operators
- Overloading the binary operator ***op*** causes automatic overloading of the assignment operator ***op*=**

This concludes our coverage of operator overloading. Notice that we have not discussed all details of this subject. You should consult a C# reference manual for full coverage.

22. Delegates

In this chapter we will discuss the concept of delegates. Seen in relation to similar, previous object-oriented programming languages (such as Java and C++) this is a new topic. The inspiration comes from functional programming where functions are *first class values*. If *x* is a first class value *x* can be passed as parameter, *x* can be returned from functions, and *x* can be part of data structures. With the introduction of delegates, methods become first class values in C#. We will explore this "exciting world be new opportunities" in the next few sections.

22.1. Delegates in C#

Lecture 6 - slide 7

The idea of a delegate in a nutshell is as follows:

> A delegate is a type the values of which consist of methods
>
> Delegates allow us to work with variables and parameters that contain methods

Thus, a delegate in C# defines a type, in the same way as a class defines a type. A delegate reflects the signature of a set of methods, not including the method names, however. A delegate is a reference type in C#. It means that values of a delegate type are accessed via references, in the same way as an object of a class always is accessed via a reference. In particular, `null` is a possible delegate value.

In Program 22.1 `NummericFunction` is the name of a new type. This is the type of functions that accept a `double` and returns a `double`. The static method `PrintTableOfFunction` takes a `NummericFunction f` as first parameter. `PrintTableOfFunction` prints a table of `f` values within a given range `[from,to]` and with a given granularity, `step`. In the `Main` method we show a number of activations of `PrintTableOfFunction`. The first three activations generate tables of the well-known functions log, sinus, and abs. Notice that these functions belong to the `NummericFunction` delegate, because they are all are functions from `double` to `double`. The last activation generates a table of the method `Cubic`, as we have defined it in Program 22.1. It is again crucial that `Cubic` is a function from `double` to `double`. If Cubic had another signature, such as `int -> int` or `double x double -> double`, it would not fit with the `NumericFunction` delegate.

```
1  using System;
2
3  public class Application {
4
5    public delegate double NumericFunction(double d);
6
7    public static void PrintTableOfFunction(NumericFunction f,
8                                            string fname,
9                                            double from, double to,
10                                           double step){
11     double d;
12
13     for(d = from; d <= to; d += step){
14      Console.WriteLine("{0,10}({1,-4:F3}) = {2}", fname, d, f(d));
15     }
16
17     Console.WriteLine();
18   }
```

```
19
20    public static double Cubic(double d){
21      return d*d*d;
22    }
23
24    public static void Main(){
25      PrintTableOfFunction(Math.Log, "log", 0.1, 5, 0.1);
26      PrintTableOfFunction(Math.Sin, "sin", 0.0, 2 * Math.PI, 0.1);
27      PrintTableOfFunction(Math.Abs, "abs", -1.0, 1.0, 0.1);
28
29      PrintTableOfFunction(Cubic, "cubic", 1.0, 5.0, 0.5);
30
31      // Equivalent to previous:
32      PrintTableOfFunction(delegate (double d){return d*d*d;},
33                           "cubic", 1.0, 5.0, 0.5);
34    }
35  }
```

Program 22.1 *A Delegate of simple numeric functions.*

In line 31 of Program 22.1 notice the *anonymous function*

```
delegate (double d){return d*d*d;}
```

The function has no name - it is anonymous. The function is equivalent with the method `Cubic` in line 20-22, apart from the fact that it has no name. It is noteworthy that we *on the fly* are able to write an expression the value of which is a method that belongs to the delegate type `NumericFunction`. In C#3.0 the notation for anonymous functions has been streamlined to that of lambda expressions. We will touch on this topic in Section 22.4. We outline the output of Program 22.1 in Listing 22.2 (only on web). We do not show all the output lines, however.

Things get even more interesting in Program 22.3. The function to watch is `Compose`. It accepts, as input parameters two numeric functions `f` and `g`, and it returns (another) numeric function. The idea is to return the function **`f`** o **`g`**. This is the function that returns f(g(x)) when it is given x as input.

Notice the expression `Compose(Cubic, Minus3)` in `Main`. This is a function that we pass as input to the `PrintTableOfFunction`, which we already have discussed. In order to examine the function `Compose(Cubic, Minus3)` we watch the program output in Listing 22.4. Please verify for yourself that `Compose(Cubic, Minus3)` is the function which subtracts 3 from its input, and thereafter calculates the cubic function on that (reduced) number.

```
1  using System;
2
3  public class Application {
4
5    public delegate double NumericFunction(double d);
6
7    public static NumericFunction Compose
8                    (NumericFunction f, NumericFunction g){
9      return delegate(double d){return f(g(d));};
10   }
11
12   public static void PrintTableOfFunction
13                   (NumericFunction f, string fname,
14                    double from, double to, double step){
15     double d;
16
17     for(d = from; d <= to; d += step){
```

```
18      Console.WriteLine("{0,35}({1,-4:F3}) = {2}", fname, d, f(d));
19    }
20
21    Console.WriteLine();
22  }
23
24  public static double Square(double d){
25    return d*d;
26  }
27
28  public static double Cubic(double d){
29    return d*d*d;
30  }
31
32  public static double Minus3(double d){
33    return d-3;
34  }
35
36  public static void Main(){
37    PrintTableOfFunction(Compose(Cubic, Minus3),
38                         "Cubic of Minus3", 0.0, 5.0, 1.0);
39
40    PrintTableOfFunction(
41      Compose(Square, delegate(double d){
42                        return d > 2 ? -d : 0;}),
43      "Square of if d>2 then -d else 0", 0.0, 5.0, 1.0);
44  }
45}
```

Program 22.3 *The static method Compose in class Application.*

```
                    Cubic of Minus3(0,000) = -27
                    Cubic of Minus3(1,000) = -8
                    Cubic of Minus3(2,000) = -1
                    Cubic of Minus3(3,000) = 0
                    Cubic of Minus3(4,000) = 1
                    Cubic of Minus3(5,000) = 8

  Square of if d>2 then -d else 0(0,000) = 0
  Square of if d>2 then -d else 0(1,000) = 0
  Square of if d>2 then -d else 0(2,000) = 0
  Square of if d>2 then -d else 0(3,000) = 9
  Square of if d>2 then -d else 0(4,000) = 16
  Square of if d>2 then -d else 0(5,000) = 25
```

Listing 22.4 *Output from the Compose delegate program.*

What we have shown above gives you the flavor of functional programming. In functional programming we often generate new functions based on existing functions, like we did with use of `Compose`.

Delegates make it possible to approach the functional programming style

Methods can be passed as parameters to, and returned as results from other methods

---

Exercise 6.3. *Finding and sorting elements in an array*

In this exercise we will work with searching and sorting in arrays. To be concrete, we work on an array of type `Point`, where `Point` is the type we have been programming in earlier exercises.

Via this exercise you are supposed to learn *how to pass a delegate to a method* such as `Find` and `Sort`. The purpose of passing a delegate to `Find` is to specify *which point we are looking for*.

Make an array of `Point` objects. You can, for instance, use this version of class Point. You can also use a version that you wrote as solution to one of the previous exercises.

Use the static method `System.Array.Find` to locate the first point in the array that satisfies the condition:

> *The sum of the x and y coordinates is (very close to) zero*

The solution involves the programming of an appropriate delegate in C#. The delegate must be a `Point` *predicate*: a method that takes a `Point` as parameter and returns a boolean value.

Next, in this exercise, sort the list of points by use of one of the static `Sort` methods in `System.Array`. Take a look at the `Sort` methods in `System.Array`. There is an overwhelming amount of these! We will use the one that takes a `Comparison` delegate, `Comparison<T>`, as the second parameter. Please find this method in your documentation browser. Why do we need to pass a `Comparison` predicate to the `Sort` method?

`Comparison<Point>` is a delegate that compares two points, say `p1` and `p2`. Pass an actual delegate parameter to `Sort` in which

```
p1 <= p2 if and only if p1.X + p1.Y <= p2.X + p2.Y
```

Please notice that a comparsion between p1 and p2 must return an integer. A negative integer means that p1 is less than p2. Zero means that p1 is equal to p2. A positive integer means that p1 is greater than p2.

Test run you program. Is your `Point` array sorted in the way you excepts?

Exercise 6.4. *How local are local variables and formal parameters?*

When we run the following program

```
using System;
public class Application {

  public delegate double NumericFunction(double d);
  static double factor = 4.0;

  public static NumericFunction MakeMultiplier(double factor){
     return delegate(double input){return input * factor;};
  }

  public static void Main(){
    NumericFunction f = MakeMultiplier(3.0);
    double input = 5.0;

    Console.WriteLine("factor = {0}", factor);
    Console.WriteLine("input = {0}", input);
    Console.WriteLine("f is a generated function which multiplies its input with
factor");
    Console.WriteLine("f(input) = input * factor = {0}", f(input));
  }
}
```

we get this output

```
factor = 4
input = 5
f is a generated function which multiplies its input with factor
f(input) = input * factor = 15
```

Explain!

22.2. Delegates that contain instance methods

Lecture 6 - slide 8

The delegates in Section 22.1 contained static methods. Static methods are activated without a receiving object. When we put an instance method *m* into a delegate object, we need to find a way to provide the receiver object of *m*. We can, in principle, provide this object as part of the activation of the delegate, or we can aggregate it together with the method itself. In C# the latter solution has been chosen.

In Section 22.1 we show a relatively trivial class `Messenger`. A messenger object stores a message of type `Message`. `Message` is a delegate, shown in **purple**. The `DoSend` method calls the method in the delegate.

```
1  using System;
2
3  public delegate void Message(string txt);
4
5  public class Messenger{
6
7    private string sender;
8    private Message message;
9
10   public Messenger(string sender){
11    this.sender = sender;
12    message = null;
13   }
14
15   public Messenger(string sender, Message aMessage){
16    this.sender = sender;
17    message = aMessage;
18   }
19
20   public void DoSend(){
21    message("Message from " + sender);
22   }
23 }
```

Program 22.5 *A Messenger class and a Message delegate.*

The class `A` is even more trivial. It just holds some state and an instance method called `MethodA`.

```
1  using System;
2
3  public class A{
4
5    private int state;
```

```
6
7     public A(int i){
8       state = i;
9     }
10
11    public void MethodA(string s){
12      Console.WriteLine("A: {0}, {1}", state, s);
13    }
14  }
```

Program 22.6 *A very simple class A with an instance method MethodA.*

In the class `Application` we create some instances of class `A`. The class `Application` is shown in Program 22.7. For now we only use one of the instances of `A`. We pass `a2.MethodA` to the `Message` (delegate) parameter of the `Messenger` constructor. With this we package both the object referred to by `a2` and the method `MethodA` together, and it now forms part of the state of the new `Message` object. When the message object receives the `DoSend` message it activates its delegate. From the output in Listing 22.8 we see that it is in fact the instance method `AMethod` in the object `a2` (with `state` equal to 2), which is called via the delegate.

```
1  using System;
2
3  public class Application{
4
5    public static void Main(){
6      A a1 = new A(1),
7        a2 = new A(2),
8        a3 = new A(3);
9
10     Messenger m = new Messenger("CS at AAU", a2.MethodA);
11
12     m.DoSend();
13   }
14
15 }
```

Program 22.7 *An Application class which accesses an instance method in class A.*

```
1 A: 2, Message from CS at AAU
```

Listing 22.8 *Output from Main of class Application.*

So now we have seen that a delegate may contain an object, which consists of a receiver together with a method to be activated on the receiver.

In Section 22.3 below we will see that this is not the whole story. A delegate may in fact contain a *list* of such receiver/method pairs.

22.3. Multivalued delegates

Lecture 6 - slide 9

The class `Messenger` in Program 22.9 is an extension of class `Messenger` in Program 22.5. The body of the method `InstallMessage` shows that it is possible to add a method to a delegate. Behind the scene, a delegate is a list of methods (and, if necessary, receiver objects). The **+** operator has been overloaded to work on

delegates. It adds a method to a delegate. Similarly, the `-` operator has been overloaded to remove a method from a delegate.

```
1  using System;
2
3  public delegate void Message(string txt);
4
5  public class Messenger{
6
7    private string sender;
8    private Message message;
9
10   public Messenger(string sender){
11    this.sender = sender;
12    message = null;
13   }
14
15   public Messenger(string sender, Message aMessage){
16    this.sender = sender;
17    message = aMessage;
18   }
19
20   public void InstallMessage(Message mes){
21    this.message += mes;
22   }
23
24   public void UnInstallMessage(Message mes){
25    this.message -= mes;
26   }
27
28   public void DoSend(){
29    message("Message from " + sender);
30   }
31 }
```

Program 22.9 *Install and UnInstall message methods in the Messenger class.*

The class `A`, which is used in Program 22.10, can be seen in Program 22.6.

In the class `Application` in Program 22.10 instantiates a number of `A` objects and a single `Messenger` object. The idea is to add and remove instance methods to the `Messenger` object, and to activate the methods in the `Messenger` object via the `DoSend` method in line 28-31 of Program 22.9.

In line 11 of Program 22.10 we install `a1.MethodA` in `m`, which already (from the `Messenger` construction) contains `a2.AMethod`. In the program output in Listing 22.11 this is revealed in the first two output lines.

Next we install `a3.AMethod` twice in `m`. At this point in time the delegate in `m` contains four methods. This is seen in the middle section of Listing 22.11.

Finally, we uninstall `a3.AMethod` and `a1.Amethod`, leaving two methods in the delegate. This is shown in the last section of output in Listing 22.11.

```
using System;

public class Application{

  public static void Main(){
    A a1 = new A(1),
      a2 = new A(2),
      a3 = new A(3);

    Messenger m = new Messenger("CS at AAU", a2.MethodA);
    m.InstallMessage(a1.MethodA);
    m.DoSend();
    Console.WriteLine();

    m.InstallMessage(a3.MethodA);
    m.InstallMessage(a3.MethodA);
    m.DoSend();
    Console.WriteLine();

    m.UnInstallMessage(a3.MethodA);
    m.UnInstallMessage(a1.MethodA);
    m.DoSend();
 }
}
```

Program 22.10 *An Application class.*

```
A: 2, Message from CS at AAU
A: 1, Message from CS at AAU

A: 2, Message from CS at AAU
A: 1, Message from CS at AAU
A: 3, Message from CS at AAU
A: 3, Message from CS at AAU

A: 2, Message from CS at AAU
A: 3, Message from CS at AAU
```

Listing 22.11 *Output from Main of class Application.*

22.4. Lambda Expressions

Lecture 6 - slide 10

A lambda expression is a value in a delegate type. Delegates were introduced in Section 22.1. The notation of lambda expression adds some extra convenience to the notation of delegates. Instead of the syntax *delegate(formal-parameters){body}* lambda expressions use the syntax *formal-parameters* **=>** *body*. **=>** is an operator in the language, see Section 6.7. It is not necessary to give the types of the formal parameters in a lambda expression. In addition, the body of a lambda expression may be an expression. In a delegate, the body must be a statement block (a command).

By the way, why is it called lambda expressions? Lambda λ is a Greek letter, like alpha α and beta α. The notion of lambda expressions come from a branch of mathematics called *lambda calculus*. In lambda calculus lambda expressions, such as *λx. x+1*, is used as a notation for functions. The particular function *λx. x+1* adds one to its argument x. Lambda expression were brought into early functional programming language, most notably Lisp. Since then, "lambda expression" has been the name of those expressions which evaluate to function values.

In Program 22.12 below we make list of five equivalent functions. The first one - line 12 - uses C# delegate notation, as already introduced in Section 22.1. The last one - line 16 - is a lambda expression written as concise as possible. The three in between - line 13, 14, and 15 - illustrate the notational transition from delegate notation to lambda notation.

```
1  using System;
2  using System.Collections.Generic;
3
4  class Program{
5
6    public delegate double NumericFunction(double d);
7
8    public static void Main(){
9
10     NumericFunction[] equivalentFunctions =
11        new NumericFunction[]{
12          delegate (double d){return d*d*d;},
13          (double d) => {return d*d*d;},
14          (double d) => d*d*d,
15          (d) => d*d*d,
16          d => d*d*d
17       };
18
19     foreach(NumericFunction nf in equivalentFunctions)
20        Console.WriteLine("NumericFunction({0}) = {1}", 5, nf(5));
21   }
22
23 }
```

Program 22.12 *Five equivalent functions - from anonymous method expressions to lambda expressions.*

In Program 22.12 notice that we are able to organize five functions in a data structure, here an array. I line 19-12 we traverse the list of functions in a **`foreach`** control structure. Each function is bound to the local name `nf`, and `nf(5)` calls a given function on the number 5.

In Listing 22.13 (only on web) we show the output of Listing 22.13. As expected, all five calls `nf(5)` return the number 125.

The items below summarize lambda expressions in relation to delegates in C#:

- The body can be a statement block or an expression
- Uses the operator **`=>`** which has low priority and is right associative
- May involve implicit inference of parameter types
- Lambda expressions serve as syntactic sugar for a delegate expression

23. Events

The event concept is central in event-driven programming. Programs with graphical user interfaces are event-driven. With the purpose of discussing events we will see a simple example of a graphical user interface at the end of this chapter.

23.1. Events

Lecture 6 - slide 12

In a program, an *event* contains some actions that must be carried out when the event is triggered

In command-driven programming, the computer prompts the user for input. When the user is prompted the program stops and waits a given program location. When a command is issued by the user, the program is continued at the mentioned location. The program will analyze the command and carry out an appropriate action.

In event-driven programming the program reacts on *what happens on the elements of the user interface*, or more generally, *what happens on some selected state of the program*. When a given event is triggered the actions that are associated with this particular event are carried out.

Inversion of control

Don't call us - we call you

The "Don't call us - we call you" idea is due to the observation that the operations called by the event mechanism is not activated explicitly by our own program. The operations triggered by events are called by the system, such as the graphical user interface framework. This is sometimes referred to as *inversion of control*.

Below, we compare operations (such as methods) and events.

- Event
  - Belongs to a class
  - Contains one or more operations, which are called when the event is *triggered*.
  - The operations in the event are *called implicitly*
- Operation
  - Belongs to a class
  - Is *called explicitly* - directly or indirectly - by other operations

In the following sections we will describe the event mechanism in C#. Fortunately, we have already made the preparations for this in Chapter 22, because an event can be modelled as a variable of a delegate type.

23.2. Events in C#

Lecture 6 - slide 13

In C# an event in some class C is a variable of a delegate type in C. Like classes, delegates are reference types. This implies that an event holds a reference to an instance of a delegate. The delegate is allocated on the heap.

> From inside some class, an event is a variable of a delegate type.
>
> From outside a class, it is only possible to add to or remove from an event.
>
> Events are intended to provide *notifications*, typically in relation to graphical user interfaces.

The following restrictions apply to events, compared to variables of delegate types:

- An event can only be activated from within the class to which the event belongs
- From outside the class it is only possible to add (with `+=`) or subtract (with `-=`) operations to an event.
    - It is not possible to 'reset' the event with an ordinary assignment

In the `System` namespace there exists a generic delegate called `EventHandler<TEVentArgs>`, which is recommended for event handling in the .NET framework. Thus, instead of programming your own delegate types of your events, it is recommended to use a delegate constructed from `EventHandler<TEVentArgs>`. The `EventHandler` delegate takes two arguments: The object which generated the event and an object which describes the event as such. The latter is assumed to be a subclass of the pre-existing class `EventArgs`. For more information about `EventHandler<TEVentArgs>` consult the documentation of the generic `EventHandler` delegate. For details on generic delegates (type parameterized delegates) see Section 43.2.

23.3. Examples of events

Lecture 6 - slide 14

In this section we will see examples of programs that make use of events.

First, in Program 23.1 we elaborate the `Die` class, which we have met several times before, see Section 10.1 , Section 12.5 , and Section 16.3.

In Program 23.1 the `Toss` operation of the `Die` class triggers a particular event in case it tosses two sixes in a row, see line 30-31.

```
1  using System;
2  using System.Collections.Generic;
3
4  public delegate void Notifier(string message);
5
6  public class Die {
7
8    private int numberOfEyes;
9    private Random randomNumberSupplier;
```

```
10   private int maxNumberOfEyes;
11   private List<int> history;
12   public event Notifier twoSixesInARow;
13
14   public int NumberOfEyes{
15      get {return numberOfEyes;}
16   }
17
18   public Die (): this(6){}
19
20   public Die (int maxNumberOfEyes){
21     randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
22     this.maxNumberOfEyes = maxNumberOfEyes;
23     numberOfEyes = randomNumberSupplier.Next(1, maxNumberOfEyes + 1);
24     history = new List<int>();
25     history.Add(numberOfEyes);
26   }
27
28   public void Toss (){
29     numberOfEyes = randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
30     history.Add(numberOfEyes);
31     if (DoWeHaveTwoSixesInARow(history))
32        twoSixesInARow("Two sixes in a row");
33   }
34
35   private bool DoWeHaveTwoSixesInARow(List<int> history){
36     int histLength = history.Count;
37     return histLength >= 2 &&
38            history[histLength-1] == 6 &&
39            history[histLength-2] == 6;
40   }
41
42   public override String ToString(){
43     return String.Format("Die[{0}]: {1}", maxNumberOfEyes, NumberOfEyes);
44   }
45 }
```

Program 23.1 *The die class with history and dieNotifier.*

In Program 23.1 `Notifier` is a delegate. Thus, `Notifier` is a type.

`twoSixesInARow` is an event - analogous to an instance variable - of type `Notifier`. Alternatively, we could have used the predefined `EventHandler` delegate (see Section 23.2) instead of `Notifier`. The event `twoSixesInARow` is public, and therefore we can add operations to this event from clients of `Die` objects. In line 9-11 of the class `diceApp`, shown in Program 23.2, we add an anonymous delegate to `d1.twoSixesInARow`, which reports the two sixes on the console.

Notice the keyword "`event`", used in declaration of variables of delegate types for event purposes. It is tempting to think of "`event`" as a modifier, which gives a slightly special semantics to a `Notifier` delegate. Technically in C#, however, `event` is not a modifier. The keyword `event` signals that we use a *strictly controlled* variable of delegate type. From outside the class, which contains the event, only addition and removal of methods/delegates are possible. The addition and removal can, inside the class, be controlled by so-called *event accessors* `add` and `remove`, which in several respect resemble `get` and `set` of properties. We will, however, not dwell on these features of C# in this material.

The predicate (boolean method) `DoWeHaveTwoSixesInARow` in line 35-40 of Program 23.1 in class `Die` determines if the die has shown two sixes in a row. This is based on the extra `history` instance variable.

Finally, the `Toss` operation may trigger the `twoSixesInARow` in line 31-32 of Program 23.1. The event is triggered in case the history tells that we have seen two sixes in a row.

```
1  using System;
2
3  class diceApp {
4
5    public static void Main(){
6
7      Die d1 = new Die();
8
9      d1.twoSixesInARow +=
10      delegate (string mes){
11        Console.WriteLine(mes);
12      };
13
14     for(int i = 1; i < 100; i++){
15       d1.Toss();
16       Console.WriteLine("{0}: {1}", i, d1.NumberOfEyes);
17     }
18
19   }
20 }
```

Program 23.2 *A client of die that reports 'two sixes in a row' via an event.*

In Program 23.3 we show the (abbreviated) output of Program 23.2. The "two sixes in a row" reporting turns out to be reported in between the two sixes. This is because the event is triggered by `Toss`, before `Toss` returns the last 6 value.

```
1  1: 6
2  2: 4
3  ...
4  32: 3
5  33: 6
6  Two sixes in a row
7  34: 6
8  Two sixes in a row
9  35: 6
10 ...
11 66: 2
12 67: 6
13 Two sixes in a row
14 68: 6
15 69: 2
16 70: 4
17 ...
18 97: 6
19 Two sixes in a row
20 98: 6
21 99: 3
```

Program 23.3 *Possible program output of the die application (abbreviated).*

We will now turn to a another example in an entirely different domain, see Program 23.4. This program constructs a graphical user interface with two buttons and a textbox, see Figure 23.1. If the user pushes the *Click Me* button, this is reported in the textbox. If the user pushes the *Erase* button, the text in the textbox is deleted.

Figure 23.1 *A graphical user interface with two buttons and a textbox.*

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

// In System:
// public delegate void EventHandler (Object sender, EventArgs e)

public class Window: Form{

  private Button b1, b2;
  private TextBox tb;

  // Constructor
  public Window (){
    this.Size=new Size(150,200);

    b1 = new Button();
    b1.Text="Click Me";
    b1.Size=new Size(100,25);
    b1.Location = new Point(25,25);
    b1.BackColor = Color.Yellow;
    b1.Click += ClickHandler;
                              // Alternatively:
                              // b1.Click+=new EventHandler(ClickHandler);
    b2 = new Button();
    b2.Text="Erase";
    b2.Size=new Size(100,25);
    b2.Location = new Point(25,55);
    b2.BackColor=Color.Green;
    b2.Click += EraseHandler;
                              // Alternatively:
                              // b2.Click+=new EventHandler(EraseHandler);
    tb = new TextBox();
    tb.Location = new Point(25,100);
    tb.Size=new Size(100,25);
    tb.BackColor=Color.White;
    tb.ReadOnly=true;
    tb.RightToLeft=RightToLeft.Yes;

    this.Controls.Add(b1);
    this.Controls.Add(b2);
    this.Controls.Add(tb);
  }

  // Event handler:
  private void ClickHandler(object obj, EventArgs ea) {
    tb.Text = "You clicked me";
  }

```

```
50    // Event handler:
51    private void EraseHandler(object obj, EventArgs ea) {
52      tb.Text = "";
53    }
54
55  }
56
57  class ButtonTest{
58
59    public static void Main(){
60      Window win = new Window();
61      Application.Run(win);
62    }
63
64  }
```

Program 23.4 *A Window with two buttons and a textbox.*

The program makes use of the already existing delegate type `System.EventHandler`. Operations in this delegate accept an `Object` and an `EventArg` parameter, and they return nothing (void).

The constructor of the class `Window` (which inherits from `Form` - a built-in class) dominates the program. In this constructor the window, aggregated by two buttons and a textbox, is built.

As emphasized in Program 23.4 we add handlers to the events `b1.Click` and `b2.Click`. We could have instantiated `EventHandler` explicitly, as shown in the comments, but the notion `b1.Click += ClickHandler` and `b2.Click += EraseHandler` is shorter and more elegant.

The two private instance methods `ClickHandler` and `EraseHandler` serve as event handlers. Notice that they conform to the signature of the `EventHandler`. (The signature is characterized by the parameter types and the return type).

Exercise 6.5. *Additional Die events*

In this exercise we add yet another method to the existing event i class `Die`, and we add another event to `Die`.

In the `Die` event example, we have a public event called `twoSixesInARow` which is triggered if a die shows two sixes in a row. In the sample client program we add an anonymous method to this event which reports the string parameter of the event on standard output.

Add yet another method to the `twoSixesInARow` event which **counts** the number of times 'two sixes in a row' appear. For this purpose we need - quite naturally - an integer variable for counting. Where should this variable be located relative to the 'counting method': Will you place the variable inside the new method, inside the `Die` class, or inside the client class of the Die?

Add a similar event called `fullHouse`, of the same type `Notifier`, which is triggered if the `Die` tosses a full house. A full house means (inspired from the rules of Yahtzee) two tosses of one kind and three tosses of another kind - in a row. For instance, the toss sequence 5 6 5 6 5 leads to a full house. Similarly, the 1 4 4 4 1 leads to a full house. The toss sequence 5 1 6 6 6 6 5 does not contain a full house sequence, and the toss sequence 6 6 6 6 6 is not a full house.

Be sure to test-drive the program and watch for triggering of both events.

24. Patterns and Techniques

In this section we will discuss the ***Observer*** design pattern. We have already introduced the idea of design patterns in Chapter 16 and we have studied one such pattern, ***Singleton***, in Section 16.3

24.1. The observer design pattern

Lecture 6 - slide 17

> The ***Observer*** is often used to ensure a *loose coupling* between an application and its user interface
>
> In general, ***Observer*** can be used whenever a set of observer objects need to be informed about state changes in a subject object

Imagine that a *weather service object* collects information about temperature, rainfall, and air pressure. When the weather conditions change significantly, a number of *weather watcher objects* - temperature watchers, rain watchers, general news watchers (newspapers and television stations) will have to be updated. See Figure 24.1.

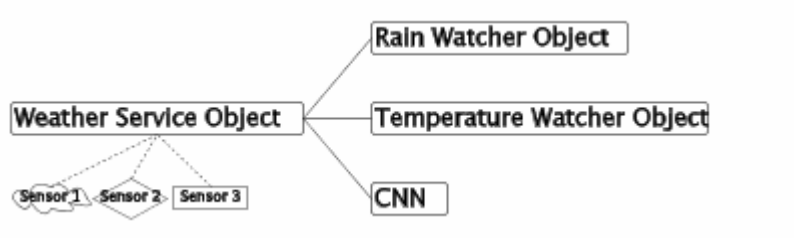

Figure 24.1 *The subject (weather service object) to the left and its three observers (weather watcher objects) to the right. The Weather Service Object get its information various sensors.*

The following questions are relevant:

1. Do the weather service object know about the detailed needs of the weather watcher objects?
2. How do we associate weather watcher objects with the weather service object?

In most naive solutions, the weather service object forwards relevant sensor observations to the weather watcher objects. The weather service object sends individual and customized messages to each weather watcher object with weather update information which is relevant for the receiver. Thus, the weather service object knows a lot about the individual needs of the watcher objects. This may work for the first two, three, or four watchers, but this approach becomes very problematic if there are many watchers: Every time a new watcher shows up we must change the weather service object.

Now let us face the second issue. In the naive solution, the weather service object will often hard wire the knowledge about watchers in the program. This is probably OK for one, two or three watchers, but it is - of course - tedious in case there are hundreds of watchers.

There is a noteworthy a solution to the problem outlined above. It is described as a design pattern, because it addresses a non-trivial solution to a frequently occurring problem. The design pattern is know as ***Observer***. The key ideas are:

1. Watcher objects *subscribe* to updates from the service object.
2. The service object *broadcasts* notifications about changes to watchers.
3. The watcher object may request details from the service object if they need to.

Below, in Program 24.1 and Program 24.2, we show the general idea/template of the ***Observer*** pattern.

```
using System.Collections;
namespace Templates.Observer {

 public class Subject {
   // Subject instance variables

   private ArrayList observers = new ArrayList();

   public void Attach(Observer o){
     observers.Add(o);
   }

   public void Detach(Observer o){
     observers.Remove(o);
   }

   public void Notify(){
     foreach(Observer o in observers) o.Update();
   }

   public SubjectState GetState(){
     return new SubjectState();
   }
 }

 public class SubjectState {
   // Selected state of the subject
 }
}
```

Program 24.1 *Template of the Subject class.*

The weather service object corresponds to an instance of class `Subject` in Program 24.1 and the watcher objects correspond to observers, as shown in Program 24.2. In Program 24.3 we illustrate how the `Observer` and `Subject` classes can be used in a client program. The programs are compilable C# programs, without any substance, however. In an appendix - Section 58.2 - we show the weather service program and how it uses the ***Observer*** pattern.

```
using System.Collections;
namespace Templates.Observer {

 public class Observer {

   private Subject mySubject;

   public Observer (Subject s){
     mySubject = s;
   }

   public void Update(){
```

```
13        // ...
14
15        SubjectState state = mySubject.GetState();
16
17        //   if (the state is interesting){
18        //      react on state change
19        //   }
20     }
21  }
22 }
```

Program 24.2 *A templates of the Observer class.*

In Program 24.3 we see that two observers, `o1` and `o2`, are attached to the subject object (line 10 and 11). The third observer `o3` is not yet attached. `o1` and `o2` hereby subscribe to updates from the subject object. Let us now assume that a mutation of the state in the subject object triggers a need for updating the observers. The following happens:

1. The subject sends a `Notify` message to itself. (In Program 24.3 the client of `Subject` and `Observer` sends the `Notify` message. This is an artificial and non-typical situation).
2. `Notify` updates each of the attached observers, by sending the parameterless `Update` message. This happens in line 18 of Program 24.1 .
3. The `Update` method in the `Observer` class asks (if necessary) what really happened in the `Subject` This is done by sending the message `GetState` back to the subject , see line 13 of Program 24.2 . Individual observers may request different information from the `Subject` . Some observers may not need to get additional information from the subject, and these observers will therefore not send a `GetState` message.
4. `GetState` returns the relevant information to the observer. The observer does whatever it finds necessary to update itself based on its new knowledge.

```
1  using Templates.Observer;
2  class Client {
3
4    public static void Main(){
5       Subject subj = new Subject();
6       Observer o1 = new Observer(subj),
7                o2 = new Observer(subj),
8                o3 = new Observer(subj);
9
10      subj.Attach(o1);  // o1 subscribes to updates from subj.
11      subj.Attach(o2);  // o2 subscribes to updates from subj.
12
13      subj.Notify();    // Following some state changes in subj
14                        // notify observers.
15   }
16 }
```

Program 24.3 *Application of the Subject and Observer classes.*

You should consult the appendix - Section 58.1 (only on web) - for a more realistic scenario in terms of the weather service and watchers.

24.2. Observer with Delegates and Events

Lecture 6 - slide 19

The Observer idea, as described in Section 24.1 can be implemented conveniently by use of events. We introduced events in Chapter 23.

According to ***Observer***, the subject has a list of observers which will have to notified when the state of the subject is updated. We can can represent the list of observers as an event. Recall from Section 23.2 that an event can contain a number of methods (all of which share a common signature described by a delegate type). Each observer adds a method to the event of the subject object. The subject notifies the observers by triggering the event.

In Program 24.4 we show a template of the `Subject` class, corresponding to Program 24.1 in Section 24.1. The event is declared in line 9. The delegate type of the event is shown in line 4. Notice that the subscription methods `AddNotifier` and `RemoveNotifier` simply adds or subtracts a method to the event. Upon notification - see line 20 in the `Notify` method - the subject triggers the event. For illustrative purposes - and in order to stay compatible with the setup in Program 24.4, we pass an instance of the subject state to the observer, see line 20 of Program 24.4. In this way there is no need for the observer to ask for it afterwards.

```
1  using System.Collections;
2  namespace Templates.Observer {
3
4   public delegate void Notification(SubjectState ss);
5
6   public class Subject {
7     // Subject instance variable
8
9     private event Notification observerNotifier;
10
11    public void AddNotifier(Notification n){
12      observerNotifier += n;
13    }
14
15    public void RemoveNotifier(Notification n){
16      observerNotifier -= n;
17    }
18
19    public void Notify(){
20      observerNotifier(new SubjectState());
21    }
22  }
23
24  public class SubjectState {
25    // Selected state of the subject
26  }
27 }
```

Program 24.4 *Template of the Subject class.*

```
1  using System.Collections;
2  namespace Templates.Observer {
3
4   public class Observer {
5
6     public Observer (){
7       // ...
8     }
9
10    public void Update(SubjectState ss){
11       //   if (the state ss is interesting){
```

```
12        //      react on state change
13        //   }
14     }
15
16  }
17 }
```

Program 24.5 *Template of the Observer class.*

In line 10-11 of Program 24.6 we see that the two observers `o1` and `o2` add their `Update` (instance) methods to the subject. This will add these methods to the event. The `Update` method of the ***Observer*** class is seen in line 10-14 of Program 24.5.

```
1  using Templates.Observer;
2  class Client {
3
4    public static void Main(){
5       Subject subj = new Subject();
6       Observer o1 = new Observer(),
7                o2 = new Observer(),
8                o3 = new Observer();
9
10      subj.AddNotifier(o1.Update);
11      subj.AddNotifier(o2.Update);
12
13      subj.Notify();
14   }
15 }
```

Program 24.6 *Application of the Subject and Observer classes.*

In an appendix - Section 58.2 - we show a version of the weather center and weather watcher program programmed with events.