13. Reference Types

Objects are accessed via references. When we create an object - by class instantiation - we obtain a reference to the new object. If we send a message to the object it is done via the reference. If the object is passed as parameter it is done via the reference. And when the object is returned from a method it is the reference to the object which is returned. We sometimes use the word *reference semantics* for all of this. Reference semantics should be seen as a contrast to value semantics. Value semantics is discussed in Chapter 14.

13.1. Reference Types

Lecture 4 - slide 2

A class is a *reference type*

Objects instantiated from classes are accessed by references

The objects are allocated on *the heap*

Instances of classes are dealt with by use of so-called *reference semantics*

Although we state that references (in C# and similar languages) correspond to pointers in C, we should be a little careful to equivalize these. In the ordinary (safe part) of C# there is no such thing as reference arithmetic, along the lines of pointer arithmetic in C. There is no address operator, and there is no dereferencing. (In an unsafe part of C# it is possible to work with pointers like in C, but we will not care about this part of the C#). References are automatically dereferenced, when it is appropriate to do so. If `r` is a reference, the expression `r.p` is used to access the property `p` in the object referenced by `r`. But the expressions `*r` and `r->p` are both illegal.

- Reference semantics:
  - Assignment, parameter passing, and **`return`** manipulates *references to objects*
- The heap:
  - The memory area where instances of classes are allocated
  - Allocation takes place when a class is instantiated
  - Deallocation takes place when the object no longer affects the program
    - In practice, when there are no references left to the object
    - By a separate thread of control called the garbage collector

13.2. Illustration of variables of reference types

Lecture 4 - slide 3

Let us now illustrate how assignments work on references. The situation shown in Figure 13.1 depicts the variables `p1` and `p2` just before we execute the assignment `p1 = p2`. The situation in the figure is established by line 1 and 2 of Program 13.1. Notice that the variables each contain a reference to a `Point` object. The variables do not contain the object themselves, but instead references to the points.

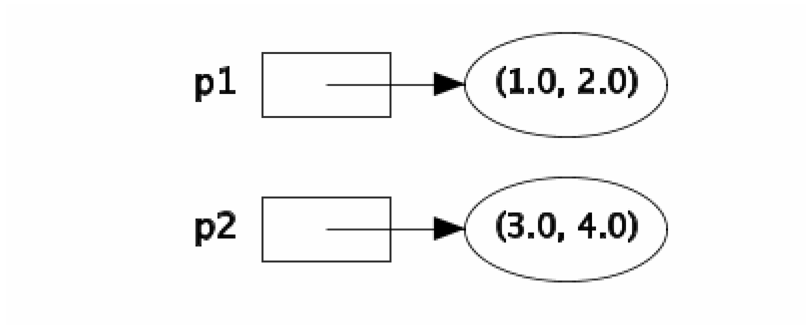

Figure 13.1 *Variables of reference types. The situation before the assignment p1 = p2.*

```
1 Point p1 = new Point(1.0, 2.0),
2       p2 = new Point(3.0, 4.0);
3
4 p1 = p2;
```

Program 13.1 *The variables p1 and p2 refer to two points.*

Following the assignment `p1 = p2` in line 3 of Program 13.1 both `p1` and `p2` reference the same `Point` object. Thus, the situation is as depicted in Figure 13.2. The `Point` (1.0, 2.0) is now inaccessible (unless referenced from other variables) and the point will disappear automatically upon the next turn of the garbage collector.

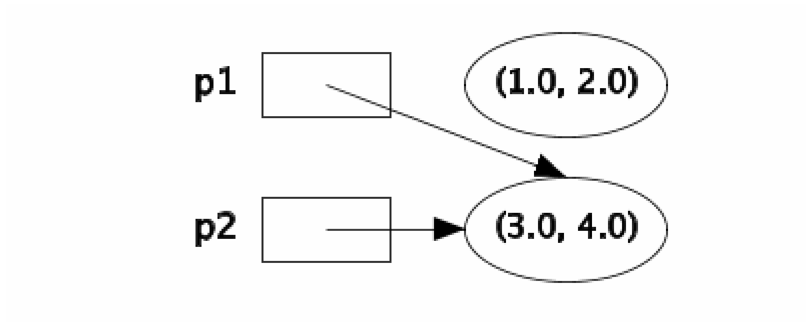

Figure 13.2 *Variables of reference types. The situation after the assignment p1 = p2.*

With the knowledge from this section you are encouraged to review the discussion of Program 12.1 in Section 12.2.

13.3. Overview of reference types in C#

Lecture 4 - slide 4

Classes are reference types in C#, but there are others as well

It is reasonable to ask which types in C# act as reference types, and which do not. Below we list the reference types in C#:

- Classes
    - Strings
    - Arrays
- Interfaces
    - Similar to classes. Contain no data. Have only signatures of methods
- Delegates
    - Delegate objects can contain one or more methods
    - Used when we deal with methods as data

We encounter interfaces in Chapter 31. Interface types contain references, and variables of interface types behave in the same way as in the example shown in Section 13.2.

A delegate is a new type, the values of which are accessed as references. We introduce delegates in Chapter 22.

Both strings and arrays are well-known, and we are used to accessing these via pointers in C. In C# - as well - arrays and strings are accessed via references.

13.4. Comparing and copying objects via references

Lecture 4 - slide 5

There are several questions that can be asked about comparing and copying objects that are accessed by references. We list some below, and we will attempt to answer the questions in the remaining parts of this section.

Do we compare references or the referenced objects?

Do we copy the reference or the referenced object?

How deep do we copy objects that reference other objects?

Let us assume, like in Program 13.1, that `p1` and `p2` are references to `Points`, where the type `Point` is defined by a class. Then the expression `p1 == p2` returns if `p1` and `p2` reference the same point. That `p1` and `p2` reference the same point means that the two involved objects are created by the same activation of `new(...)`. In many context, we say that `p1` and `p2` are *identical*. (Identical objects and object identity is discussed in Section 11.14). Relative to Figure 13.1 the value of `p1 == p2` is false. Relative to Figure 13.2 the value of `p1 == p2` is true. The expression `p1 == p2` compares the locations (addresses) to which p1 and p2 refer. The expression does <u>not</u> compare the instance variables of the points referred to by `p1` and `p2`.

In the same way, the assignment `p1 = p2` manipulates only the references. We have already seen that in Section 13.2. The assignment `p1 = p2` does not, in any way, copy the object referenced by `p2`.

Above, we have explained *reference comparison and assignment*. It makes sense to have *shallow* and *deep* variations of these. This can be summarized as follows:

- Comparing
    - Reference comparison
        - Compares if two references refers to objects created by the same execution of **`new`**
    - Shallow and deep comparison
        - Pair-wise comparison of fields - at varying levels
- Copying:
    - Reference copying
    - Shallow or deep copying
        - Is also known as *cloning*
        - Somehow supported by the method **`MemberwiseClone`** in **`System.Object`**

Even in the case where `p1 == p2` is false (i.e., `p1` and `p2` are not identical) it makes sense to claim that `p1` and `p2` are equal in some sense. It may, for instance, be the case that all instance variables are pair-wise equal. But what does it mean for the instance variables to be pair-wise equal? In case the instance variables are references we are back to the original question, and we can therefore apply recursion in our reasoning about equality. We talk about *shallow equality* if we apply (fall back to) reference equality at the second level. If we do not apply reference equality at any level we talk about *deep equality*. If `p1` and `p2` are deep equal the graph structures they reference are structural identical (isomorphic).

If p1 and p2 are reference equal they are also shallow equal. And if p1 and p2 are shallow equal they are also deep equal. The inverse propositions are not necessarily true, of course.

If you got the idea of the different kinds of comparison, you can immediately use this insight for copying as well. Let us describe this very briefly. The assignment `p1 = p2` just copies one reference. We may ask for a shallow copy of `p2` by copying value fields and by assigning corresponding reference fields to each other. And we may ask for a deep copy by not using reference copying at any level. (This is not exactly true if there is more than one reference to a given object - please consider!)

In case you need shallow or deep copying you should program such operations yourself. In general, various kinds of copying depend deeply on the type of the object. C# supports a shallow clone operation, but you must explicitly 'enable it'. How this is done is discussed in Section 32.7.

An assignment of the form **`var = obj1`** copies a reference A comparison of the form **`obj1 == obj2`** compares references (unless overloaded)

13.5. Equality in C#

Lecture 4 - slide 6

In this section we review the different equality operations in C#. All methods mentioned in this section belong the class `Object`, see Section 28.3. We only care about reference types in this section, because the enclosing chapter is about reference types. Equality among 'objects' that belong to value types is an different story.

- **o1.Equals(o2)** - *equality*
  - By default, true if **o1** and **o2** are created by execution of the same **new**
  - Can be redefined in a particular class
- **Object.ReferenceEquals(o1, o2)** - *identity*
  - True if both **o1** and **o2** are `null`, or if they are created by execution of the same **new**
  - Static - cannot be redefined.
- **Object.Equals(o1, o2)**
  - True if **Object.ReferenceEquals(o1, o2)**, or if **o1.Equals(o2)**
- **o1 == o2**
  - True if both **o1** and **o2** are null, or if they are created by execution of the same **new**
  - An overloadable operator

Notice that in case we need a type-dependent comparison we redefine the `Equals` (in the first item above). `Equals` is typically redefined if we wish to implement a value-like comparison of two objects, as opposed to the default reference comparison. (In a value-like comparison we compare pairs of fields from the two objects). It is not easy to redefine `Equals` correctly. We discuss how it should be done in Section 28.16.

`ReferenceEquals` is a static method. It must therefore be activated by the form `Object.ReferenceEquals(o1, o2)`. If you, for some reason, redefine the == operator as well as the `Equals` instance method - both of which per default are reference equality operations - the static `ReferenceEquals` comes in handy if you need to compare references to objects. Alternatively, you will have to cast one of operands to type `Object` before you use ==.

The static `Equals` method is primarily justified because it allows one or both of the parameters `o1` or `o2` to be null. In the non-static `Equals` methods, it will cause an exception if `o1` is null. Notice that redefinition of the `Equals` instance method affects the static `Equals` method.

In C# it is allowed to overload the == operator. Typically, == is overloaded to obtain some kind of shallow comparison, see Section 13.4. If the == operator is overloaded you should also redefine the `Equals` method, such that `o1 == o2` and `o1.Equals(o2)` have the same value (whenever `o1` is not `null`).

It is worth pointing out that the meaning of `o1 == o2` is resolved staticially, because operator overloading (see Chapter 21) is a static issue in C#. In contrast, `o1.Equals(o2)` is resolved dynamically, because the instance method `Equals` is a virtual method (see Section 28.14) in class `Object`. This affects both flexibility (where `Equals` is the winner) and efficiency (where == is the winner). Exercise 4.1 is related to theses observations.

It is worthwhile and recommended to read about equality in the C# documentation of `Equals` in the `System` namespace. Let us also point out that there exists a couple of interfaces that involve equality, most directly `IEquality` (see Section 42.9), but indirectly also `Icomparable` (see Section 42.8).

Redefinition of equality operators and methods: Recommendations. FOCUS BOX 13.1

As above, we assume that we deal with reference types. If you do not redefine the `Equals` instance method nor the == operator, both of them denote reference equality.

If it is *natural and important* that equality between objects of your class should rely on the data contents (instance variables) of your class, rather than the referenced locations of the involved objects, you should redefine the `Equals` instance method. Follow the guidelines in Section 28.16. As part of this, remember that equality should be *reflexive* (`x.Equals(x)`), *symmetric* (`x.Equals(y)` implies that `y.Equals(x)`), and *transitive* (`x.Equals(y)` and `y.Equals(z)` implies that `x.Equals(z)`).

In general, you are not recommended to overload (redefine) the == operator. Most programmers with a C background will be surprised if `x == y` (for references or pointers) does not compare the references in `x` and `y`. If you overload the `Equals` instance method, you most likely do not want to touch the == operator. Thus, == will remain as the reference equality operator.

If - against these recommendations - you overload the == operator, you should make sure that the meaning (semantics) of == and `Equals` are the same. This can, for instance, be obtained by implementing `Equals` by means of ==.

It would be *tremendously confusing* to have two different meanings of == and `Equals`, both of which differ from the meaning of `ReferenceEquals`.

Exercise 4.1. *Equality of value types and reference types*

Take a close look at the following program which uses the == operator on integers.

```
using System;

class WorderingAboutEquality{

  public static void Main(){
    int i = 5,
        j = 5;

    object m = 5,
           n = 5;

    Console.WriteLine(i == j);
    Console.WriteLine(m == n);
  }

}
```

Predict the result of the program. Compile and run the program, and explain the results. Were your predictions correct?

14. Value Types

Values - in value types - are not accessed via references. In the safe part of C# it is not possible to access such values via references. Variables of value types contain their values (and not references to their values). This implies that values are allocated on the *method stack*, and the creation and deletion of such values are easier for the programmer to deal with than objects on the *heap*.

The numeric types, char, boolean and enumeration types are value types in C#. In addition, structs are value types in C#. (The numeric types, `char`, and `boolean` are - in fact - defined as structs in C#).

We will normally use the word "*object*" with the meaning "*instance of a class*". With this meaning, objects are accessed by references. But in some sense, values (of value types) are also objects in C#. Both value types and reference types inherit from the class `Object`. Thus, class `Object` is the common superclass of both reference types and value types. See Section 28.2 for additional clarification of this issue.

In order to avoid unnecessary confusion, we will - unless stated explicitly - devote the word "object" to instances of classes.

14.1. Value types

Lecture 4 - slide 8

In this section we introduce the term *value semantics*.

A variable of value type contains its value

The values are allocated on *the method stack* or within objects on the heap

Variables of value types are dealt with by use of so-called *value semantics*

Use of value types simplifies the management of short-lived data

- Value semantics
  - Assignment, call-by-value parameter passing, and **`return`** copy entire values
- The method stack
  - The memory area where short-lived data is allocated
    - Parameters and local variables
  - Allocation takes place when an operation, such as a method, is called
  - Deallocation takes place when the operation returns

Data on the method stack corresponds to variables of storage class auto in C programming.

14.2. Illustration of variables of value types

Lecture 4 - slide 9

Assume that the type Point is a value type

We will now demonstrate how value semantics works in relation to assignments.

We will assume that the type `Point` is a value type. In C# it will be programmed as a struct. We show `Point` defined as a struct in Section 14.3.

In Figure 14.1 we show two variables, `p1` and `p2`, that contain `Point` values. The situation in Figure 14.1 can, for instance, be established by the initializers associated with the declarations of `p1` and `p2` in Program 14.1 . The assignment `p1 = p2`, also shown in Program 14.1, establishes the situation in Figure 14.2.

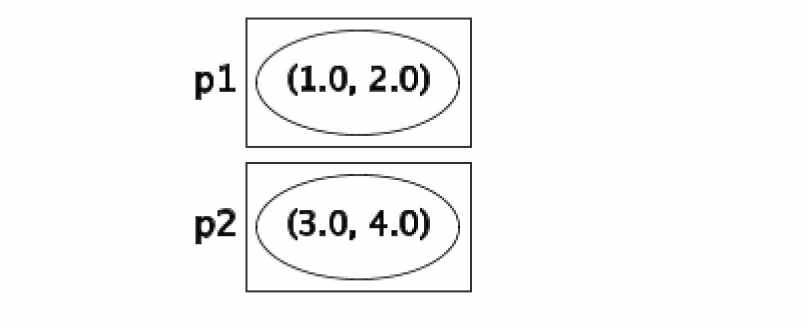

Figure 14.1 *Variables of value types. The situation before the assignment p1 = p2.*

```
Point p1 = new Point(1.0, 2.0),
      p2 = new Point(3.0, 4.0);

p1 = p2;
```

Program 14.1 *The variables p1 and p2 refer to two points.*

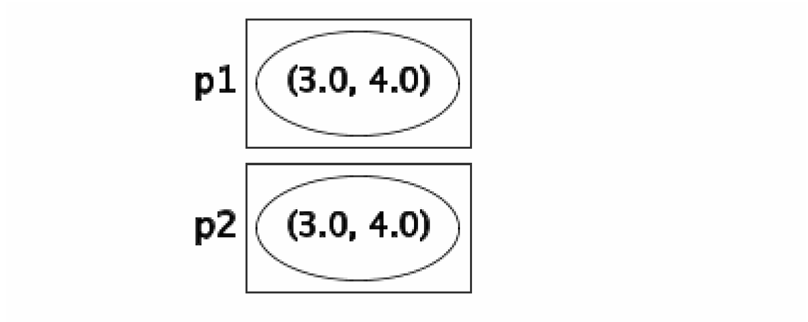

Figure 14.2 *Variables of value types. The situation after the assignment p1 = p2.*

The thing to notice is that the assignment `p1 = p2` copies the value contained in `p2` into the variable `p1`. The coping process can be implemented as a bitwise copy, and therefore it is relatively efficient.

The equality operator `p1 == p2` compares the values in `p1` and `p2` (bitwise comparison). Let us also observe that `p1.Equals(p2)` has the same boolean value as `p1 == p2` when the type of `p1` and `p2` is a value type.

The observations about assignments from above can also be used directly on call-by-value parameter passing. Call-by-value parameter passing is - in reality - assignment of the actual parameter value to the corresponding formal parameter.

As a contrast to the description of value assignment, please see Section 13.2 where we showed what happens if `p1` and `p2` are declared as classes (of reference types). Notice that `p1 = p2`, in case `p1` and `p2` contain references, is likely to be even more efficient than the value assignment discussed above.

14.3. Structs in C#

Lecture 4 - slide 10

In this section we will study two C# types, which we program as structs. The two types become value types. The first, `Point`, is already well-known. See Program 11.2. The other, `Card`, is also one of our recurring examples. In Program 12.7 we programmed `Card` as a class.

In Program 14.2 we show a simple `Point` struct. In this version the data representation is private. Notice also the constructor. The constructor is used to initialize a new point. In addition there are three methods `GetX`, `GetY`, and `Move`. When we learn more about C# we will most likely program `GetX` and `GetY` as properties, see Section 18.1. We may also chose to program `Move` in a functional style, such that the struct `Point` becomes immutable. Immutable types are discussed in Section 14.7.

Like in classes, it is always recommended that you program one or more constructors in a struct. It cannot be a parameterless constructor, however. See Section 14.4 for details on structure initialization.

The usage of struct `Point` has already been illustrated above, see Program 14.1 in Section 14.2.

```
using System;

public struct Point {
  private double x, y;

  public Point(double x, double y){
   this.x = x; this.y = y;
  }

  public double Getx (){
    return x;
  }

  public double Gety (){
    return y;
  }

  public void Move(double dx, double dy){
    x += dx; y += dy;
  }

  public override string ToString(){
    return "Point: " + "(" + x + "," + y + ")" + ".";
  }
}
```

Program 14.2 *Struct Point.*

In Program 14.3 we show the struct `Card`. Struct `Card` represents a playing card. It uses enumeration types for card suites and card values. The playing card has private fields in line 11 and 12, as we will expect. The struct is well-equipped with constructors for flexible initialization of new playing cards. The method `Color` calculates a card color from its suite and value. The method returns a value of the pre-existing type

System.Drawing.Color. Interesting enough in this context, System.Drawing.Color is also a struct. We use the fully qualified name of class Color in the namespace System.Drawing in order not to get a conflict with the Color member in struct Card.

Finally, the usual ToString (overridden from class Object) allows us to print playing cards. This is, of course, very convenient when we write small programs that uses struct Card.

```
using System;

public enum CardSuite:byte
          {Spades, Hearts, Clubs, Diamonds };
public enum CardValue: byte
          {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 struct Card {
  private CardSuite suite;
  private CardValue value;

  public Card(CardSuite suite, CardValue value){
   this.suite = suite;
   this.value = value;
  }

  public Card(CardSuite suite, int value){
    this.suite = suite;
    this.value = (CardValue)value;
  }

  public CardSuite Suite(){
    return this.suite;
  }

  public CardValue Value (){
    return this.value;
  }

  public System.Drawing.Color Color (){
   System.Drawing.Color result;
   if (suite == CardSuite.Spades || suite == CardSuite.Clubs)
     result = System.Drawing.Color.Black;
   else
     result = System.Drawing.Color.Red;
   return result;
  }

  public override String ToString(){
    return String.Format("Suite:{0}, Value:{1}, Color:{2}",
                         suite, value, Color().ToString());
  }
}
```

Program 14.3 *Struct Card.*

A simple client of Card, which declares and constructs three playing cards, is shown in Program 14.4. The card in c1 is copied to c4. Finally, all cards are printed with WriteLine, which internally uses the programmed ToString method in struct Card.

```
using System;

public class PlayingCardClient{

  public static void Main(){
    Card c1 = new Card(CardSuite.Spades, CardValue.King),
         c2 = new Card(CardSuite.Hearts, 1),
         c3 = new Card(CardSuite.Diamonds, 13),
         c4;

    c4 = c1;  // Copies c1 into c4

    Console.WriteLine(c1);
    Console.WriteLine(c2);
    Console.WriteLine(c3);
    Console.WriteLine(c4);
  }

}
```

Program 14.4 *A client of struct Card.*

Structs are typically used for aggregation and encapsulation of *a few values*, which we want to treat as a value itself, and for which we wish to apply value semantics

In the `System` namespace, the types **`DateTime`** and **`TimeSpan`** are programmed as structs

Very large structs, which encapsulates many data members, are not often seen. It is most attractive to use structs for small bundles of data, because structs are copied back and forth when we operate on them.

It is instructive to study the interfaces of `System.DateTime` and `System.TimeSpan`, which both are programmed as structs in the C# standard library.

14.4. Structs and Initialization

Lecture 4 - slide 11

There are some peculiar rules about initialization of struct values, at least if compared to initialization of class instances. We will review these peculiarities in this section.

Program 14.5 shows that initializers, such as '`= 5`' and '`= 6.6`' cannot be used with structs. The designers of C# insist that the default value of a struct is predictable, as formed by the default values of the types of the instance variables `a` and `b`.

```
/* Right, Wrong */
using System;

// Error:
// Cannot have instance field initializers in structs.
public struct StructOne{
  int a = 5;
  double b = 6.6;
}

// OK:
// Fields in structs are initialized to default values.
public struct StructTwo{
  int a;
  double b;
}
```

Program 14.5 *Fields in structs cannot have initializers.*

Program 14.6 shows that we cannot program parameterless constructors in a struct. This would overwrite the preexisting default constructor, which initializes all fields to their default values. The designers of C# wish to control the default constructor of structs. The default constructor of a struct therefore always initializes instance variables to their default values. Our own struct constructors should all have at least one parameter.

```
/* Right, Wrong */
using System;

// Error:
// Structs cannot contain explicit parameterless constructors.
public struct StructThree{
  int a;
  double b;

  public StructThree(){
    a = 1;
    b = 2.2;
  }
}

// OK:
// We can program a constructor with parameters.
// The implicit parameterless constructor is still available.
public struct StructFour{
  int a;
  double b;

  public StructFour(int a, double b){
    this.a = a;
    this.b = b;
  }
}
```

Program 14.6 *An explicit parameterless constructor is not allowed.*

14.5. Structs versus classes

Lecture 4 - slide 12

In order to summarize structs in relation to classes we provide the following comparison:

Classes	**Structs**
Reference type	Value type
Used with dynamic instantiation	Used with static instantiation
Ancestors of class `Object`	Ancestors of class `Object`
Can be extended by inheritance	Cannot be extended by inheritance
Can implement one or more interfaces	Can implement one or more interfaces
Can initialize fields with initializers	Cannot initialize fields with initializers
Can have a parameterless constructor	Cannot have a parameterless constructor

14.6. Examples of mutable structs in C#

Lecture 4 - slide 13

Structs are often used for immutable objects. (Here we use 'object' in a loose sense, covering both struct values and class instances). An object is immutable if its state cannot be changed once the object has been initialized. Recall that strings in C# are immutable.

We start by studying mutable structs, and hereby we seek motivation for dealing with immutable structs.

Please take a new look at struct `Point` in Program 14.2 from Section 14.3. In particular, focus your attention on the `Move` method. A call such as `p.Move(7.0, 8.0)` will change the state of point `p`. We say that the point `p` has been mutated.

In Program 14.7, which is a client of struct `Point` from Program 14.2, the point `p1` is moved twice. The program output in Listing 14.8 (only on web) is as expected.

```
using System;

public class Application{

  public static void Main(){
    Point p1 = new Point(1.0, 2.0);

    p1.Move(3.0, 4.0);     // p1 has moved to (4.0, 6.0)
    p1.Move(5.0, 6.0);     // p1 has moved to (9.0, 12.0)

    Console.WriteLine("{0}", p1);
  }

}
```

Program 14.7   *Moving a point by mutation.*

The struct in Program 14.9 is similar to Program 14.2. The difference is that `Move` in Program 14.9 returns a point, namely the current point, denoted by **`this`**. But - as shown in Program 14.10 this causes troubles in some situations. Following the program we will explain the reason.

```
using System;

public struct Point {
  private double x, y;

  public Point(double x, double y){
    this.x = x; this.y = y;
  }

  public double Getx (){
    return x;
  }

  public double Gety (){
    return y;
  }

  public  Point  Move(double dx, double dy){
    x += dx; y += dy;
    return this;  // returns a copy of the current object
  }

  public override string ToString(){
    return "Point: " + "(" + x + "," + y + ")" + ".";
  }
}
```

Program 14.9 *The struct Point - mutable, where move returns a Point.*

In Program 14.10 the expression `p1.Move(3.0, 4.0).Move(5.0, 6.0)` is parsed as `(p1.Move(3.0, 4.0)).Move(5.0, 6.0)` due the left associativity of the dot operator. So `p1` is first moved by 3.0 and 4.0 to (4, 6). `Move` returns **a new copy of** the point (4, 6). (This observation is important). This new copy of the point is an anonymous point, because it it is not contained in any variable. The anonymous point is then moved to (9.0, 12.0). In line 9 of Program 14.10 we print `p1`, which - as argued - is located at (4, 6). The program output shown in Listing 14.11 confirms our observations.

```
1  using System;
2  
3  public class Application{
4  
5     public static void Main(){
6      Point p1 = new Point(1.0, 2.0);
7  
8      p1.Move(3.0, 4.0).Move(5.0, 6.0);
9      Console.WriteLine("{0}", p1);      // Where is p1 located?
10   }
11 }
```

Program 14.10 *Application the struct Point - Cascaded moving.*

```
Point: (4,6).
```

Listing 14.11 *Output from the application.*

The state of affairs in Program 14.10 is not satisfactory. We have mixed imperative and functional programming in an unfortunate way. In the following section we will make another version of `Move` that works as expected when used in the cascading manner, such as in the expression `p1.Move(3.0, 4.0).Move(5.0, 6.0)`. The new version will be programmed in a functional way, and it will illustrate use of immutable structs.

14.7. Examples of immutable structs in C#

Lecture 4 - slide 14

As an alternative to `Move` in Program 14.9 we can program `Move` in such a way that an expression like `p.Move(7.0, 8.0)` returns a new point, different from the point in `p`. The new point is displaced 7.0 in the x direction and 8.0 in the y direction relative to the point in `p`. The state of `p` is not changed by `Move`. We typically want to get hold on the new point in an assignment, such as in

```
q = p.Move(7.0, 8.0);
```

Program 14.12 shows yet another version of struct `Point`, in which `Move` constructs and returns a new point. In this version `Point` is immutable. Once constructed we never change the coordinates of a point. This is signalled by making the instance variables `x` and `y` readonly, see line 4.

Notice the difference between `Move` in Program 14.12 and `Move` in Program 14.9.

```
1  using System;
2  
3  public struct Point {
4    private readonly double x, y;
5  
6    public Point(double x, double y){
7      this.x = x; this.y = y;
8    }
9  
10   public double Getx (){
11     return x;
12   }
13 
14   public double Gety (){
15     return y;
16   }
17 
18   public Point Move(double dx, double dy){
19     return new Point(x+dx, y+dy);
20   }
21 
22   public override string ToString(){
23     return "Point: " + "(" + x + "," + y + ")" + ".";
24   }
25 }
```

Program 14.12 *The struct Point - immutable.*

In Program 14.13 we show the counterpart to Program 14.10 and Program 14.7.

The expression `p1.Move(3.0, 4.0).Move(5.0, 6.0)` now does the following:

1. `p1.Move(3.0, 4.0)` returns a copy of the point in `p1`. The copy is located in (4,6).
2. The point in (4,6) is moved to (9,12) by the second call to `Move`. This creates yet another point.

The 'yet another point' is finally copied into the variable `p2`.

```
using System;

public class Application{

   public static void Main(){
     Point p1 = new Point(1.0, 2.0),
           p2;

     p2 = p1.Move(3.0, 4.0).Move(5.0, 6.0);
     Console.WriteLine("{0} {1}", p1, p2);
   }

}
```

Program 14.13 *Application the struct Point - immutable.*

As shown in Listing 14.14 the original point in `p1` is not altered. The point, which finally is copied into `p2`, is located as expected.

```
Point: (1,2). Point: (9,12).
```

Listing 14.14 *Output from the application.*

There is a misfit between mutable datatypes and use of value semantics

It is recommended to use structs in C# together with a functional programming style

The deep insight of all this is that we should strive for a functional programming style when we deal with structs. Structs are born to obey value semantics. This does not fit with the 'imperative point mutation' idea, as exemplified in Program 14.9 and Program 14.10. Use the style in Program 14.12 and Program 14.13 instead.

In this and the previous section I have benefited from Sestoft's and Hansen's explanations and examples from the book *C# Precisely*.

Exercise 4.2. *Are playing cards and dice immutable?*

Evaluate and discuss the classes `Die` and `Card` with respect to mutability. (If you access this exercise from the web edition there are direct links to the relevant versions of class `Die` and class `Card`).

Make sure that you understand what mutability means relative to the concrete code. Explain it to your fellow programmers!

More specific, can you argue for or against the claim that a `Die` instance/value should be mutable?

And similarly, can you argue for or against the claim that a `Card` instance/value should be mutable?

Why is it natural to use structs for immutable objects and classes for mutable objects? Please compare your findings with the remarks in 'the solution' when it is released.

14.8. Boxing and Unboxing

Lecture 4 - slide 15

C# has a uniform type system in the sense that both values types and reference types are compatible. Conceptually, the compatibility is ensured by the fact that both value types and reference types are derived from the class `Object`. See Section 28.2. Operationally, the compatibility is ensured by the boxing of value types. This will be the theme in this section.

Boxing involves a wrapping of a value in an object of the class **`Object`**

Unboxing involves an unwrapping of a value from an object

- Boxing
  - Done as an *implicit* type conversion
  - Involves allocation of a new object on the heap and copying of the value into that object
- Unboxing
  - Done *explicitly* via a type cast
  - Involves extraction of the value of a box

Boxing takes place when a simple value or a struct is bound to a variable or a parameter of reference type. This is, for instance, the case if an integer value is passed to a parameter of type `Object` in a method.

When a value is boxed it is embedded in an object on the heap, together with information about the type of the value. If the boxed value (an object) is unboxed it can therefore be checked if the unboxing makes sense.

In Program 14.15 we first illustrate boxing of an integer `i` and a boolean `b` in line 8 and 9. The boxing is done implicitly. Next follows unboxing of the already boxed values in line 11 and 12. Unboxing must be done explicitly. Unboxing is accomplished by casts, both in the assignments to `j` and `c`, respectively, and in the context of the arithmetic and logical expressions. Line 14 and 15 illustrate attempts to do unboxing without casts. This is illegal, and the compiler finds out.

We are able to print both objects and values in the final `WriteLine` of Program 14.15. This is because the method `ToString` uses the type information of a boxed value to provide for natural generation of a printable string.

```
1  using System;
2  
3  public class BoxingTest{
4    public static void Main(){
5      int i = 15, j, k;
6  
7      bool b = false, c, d;
8      Object obj1 = i,    // boxing of the value of i
9             obj2 = b;    // boxing of the value of b
10 
11     j = (int) obj1;     // unboxing obj1
12     c = (bool) obj2;    // unboxing obj2
13 
14 //  k = i + obj1;     // Compilation error
15 //  d = b && obj2;    // Compilation error
16 
17     k = i + (int)obj1;
18     d = b && (bool)obj2;
19 
20     Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}",
21                       i, obj1, b, obj2, j, c, k, d);
22 
23   }
24 }
```

Program 14.15 *A program that illustrates boxing and unboxing.*

The output of the program is shown in Listing 14.16.

```
15, 15, False, False, 15, False, 30, False
```

Listing 14.16 *Output of the boxing and unboxing program.*

14.9. Nullable types

Lecture 4 - slide 16

A variable of reference type **can** be null

A variable of value type **cannot** be null

A variable of *nullable value type* **can** be null

Nullable types provide a solution to the following desire:

> All values of a value type **t** (such as `int`) 'are in use'. In some programs we wish to have a distinguished value in **t** which stands for 'no value'.

When we use reference types we use the distinguished `null` value for such purposes. However, when we program with value types this is not possible. Therefore the concept of *nullable types* has been invented. It allows a variable of a value type to have the (distinguished) `null` value.

Before we see how nullable types are expressed in C# we will take a look at a motivating example, programmed without use of nullable types. The full details of the example are available in the web-version of the material. In Program 14.17 (only on web) we program a simple integer sequence class, which represents an ordered sequence of integer values. We provide this type with `Min` and `Max` operations. The problem is which value to return from `Min` and `Max` in case the sequence is empty. In Program 14.17 we return -1, but this is a bad solution because -1 may very well be the minimum or the maximum number in the sequence. Please make sure that you understand the problem in Program 14.17 before you proceed.

In Program 14.18 we show another version of class `IntSequence`. In this solution, the methods `Min` and `Max` return a value of type **`int?`**. **`int?`** means a nullable integer type. Thus, the value `null` is a legal value in **`int?`**. This is exactly what we need because `Min` and `Max` are now able to signal that there is no minimum/maximum value in an empty sequence.

```
public class IntSequence {

  private int[] sequence;

  public IntSequence(params int[] elements){
    sequence = new int[elements.Length];
    for(int i = 0; i < elements.Length; i++){
      sequence[i] = elements[i];
    }
  }

  public int? Min(){
    int theMinimum;
    if (sequence.Length == 0)
      return null;
    else {
      theMinimum = sequence[0];
      foreach(int e in sequence)
        if (e < theMinimum)
           theMinimum = e;
    }
    return theMinimum;
  }

  public int? Max(){
    int theMaximum;
    if (sequence.Length == 0)
      return null;
    else {
      theMaximum = sequence[0];
      foreach(int e in sequence)
        if (e > theMaximum)
           theMaximum = e;
    }
    return theMaximum;
  }

  // Other useful sequence methods

}
```

Program 14.18 *An integer sequence with Min and Max operations - with int?.*

In Program 14.19 we show an application of class `IntSequence`, where we illustrate both empty and non-empty sequences. Notice the use of the property `HasValue` in line 14. The property `HasValue` can be applied on a value of a nullable type. The output of the program is shown in Listing 14.20 (only on web).

```
1  using System;
2  
3  class IntSequenceClient{
4  
5    public static void Main(){
6      IntSequence is1 = new IntSequence(-5, -1, 7, -8, 13),
7                  is2 = new IntSequence();
8  
9      ReportMinMax(is1);
10     ReportMinMax(is2);
11   }
12 
13   public static void ReportMinMax(IntSequence iseq){
14     if (iseq.Min().HasValue && iseq.Max().HasValue)
15       Console.WriteLine("Min: {0}. Max: {1}",
16                          iseq.Min(), iseq.Max());
17     else
18       Console.WriteLine("Int sequence is empty");
19   }
20 
21 }
```

Program 14.19 *A client of IntSequence.*

Let us now summarize the important properties of nullable types in C#:

- Many operators are *lifted* such that they also are applicable on nullable types
- An implicit conversion can take place from **`t`** to **`t?`**
- An explicit conversion (a cast) is needed from **`t?`** to **`t`**

The observations about implicit conversion from a non-nullable type **`t`** to its nullable type **`t?`** is as expected. A value in a narrow type can be converted to a value in a broader type. The other way around requires an explicit cast.

Only value types can be nullable. It is therefore possible to have nullable struct types. It is only possible to built a nullable type on a non-nullable type. Therefore, the types **`t??`**, **`t???`**, etc. are undefined in C#.

A nullable type **`t?`** is itself a value type. It might be tempting to consider a value *v* of **`t?`** as a boxing of *v* (see Section 14.8). This is, however, not a correct interpretation. A boxed value belongs to a reference type. A value in **`t?`** belongs to a value type.

The nullable type **`t?`** is syntactic sugar for the type **`Nullable<t>`** for some given value type **`t`**. **`Nullable<t>`** is a generic struct, which we discuss briefly in Section 42.7.

The type **`bool?`** has three values: *true*, *false*, and *null*. The operators `&` and `|` have been lifted to deal with the *null* value. In addition, conditional and iterative control structures allow control expressions of type **`bool?`**. In these control structures *null* counts as *false*.

The *null-coalescing C# operator* **`??`** is convenient when we work with nullable types. The expression `x ?? y` is a shortcut of `x != null ? x : y`. The **`??`** operator can be used to provide default values of variables of nullable types. In the context of

```
int? i = null,
     j = 7;
```

the expression `i ?? 5` returns 5, but `j ?? 5` returns 7. The **`??`** operator can also be used on reference types!

15. Organization of C# Programs

This chapter is of a different nature than the previous chapters.

At this point you are supposed to be able to program simple classes, like the `Die` class in Program 10.1, the `Point` class in Program 11.2, and the `BankAccount` class in Program 11.5. Eventually, it will be necessary to care about how classes are organized in relation to each other. We chose to cover C# program organization now. In case you are not motivated for these issues, you can skip the chapter at this point in time. But you are advised to come back to it before you start writing large C# programs.

If you want to read more about the organization of C# programs, you are recommended to study chapter 16 of C# Language Specification [ECMA-334].

We show a lot of examples in this chapter. In the web edition, all examples are present. In the paper edition, only the most fundamental examples appear. Therefore, if you want to understand all the details of this chapter, read the web edition.

15.1. Program Organization

Lecture 4 - slide 18

The structure and organization of a C# program is radically different from the structure and organization of both C programs and Java programs. Below we emphasize some important observations about the organization of C# programs.

- C# programs are organized in namespaces
    - Namespace can contain types and - recursively - other namespaces
- Namespaces (and classes) in relation to files:
    - One of more namespaces in a single file
    - A single namespace in several files
    - A single class in several files - partial classes
- The mutual order of declarations is not significant in C#.
    - No *declaration before use*.
- When compiled, C# programs are organized in assemblies
    - **`.exe`** and **`.dll`** files

As noticed above, a single namespace can be spread out on several source files. In Section 11.12 we have also seen that a single class - called a *partial class* - can be defined in two or more source files.

15.2. Examples of Program Organization

Lecture 4 - slide 19

In the majority of the small programs, which we present in this material, namespaces do not appear explicitly. Most of the programs we have shown until now in this material follow the pattern of Program 15.1.

```
// The most simple organization
// A class located directly in the global namespace
// In source file ex.cs

using System;

public class C {

  public static void Main(){
    Console.WriteLine("The start of the program");
  }

}
```

Program 15.1 *A single class in the anonymous default namespace.*

In Program 15.1 the class `C` is a member of the (implicitly stated) *global name space*. The compilation of the program in Program 15.1 can be done as shown in Listing 15.2 (only on web).

Below, in Program 15.3 the namespaces `N1` and `N2` are members of the global name space. `N2` contains a nested namespace called N3.

You should use namespaces to group classes that somehow belong together, either conceptually or according the architecture of the software you are creating. Namespaces are also useful if you have identically named types (such as two classes with the same name) that should coexist. In that case, place the conflicting types in different namespaces, and be sure to use the involved namespaces with qualified access - "namespace dotting". Use of several namespaces, such as `N1`, `N2`, and `N3` in Program 15.3 is, in general, relevant only in large programs with many types.

```
// Several namespaces, including nested namespaces.
// In source file ex.cs

namespace N1 {
  public class C1{};
}


namespace N2 {
  internal class C2{};
  public class C3{};

  namespace N3 {
     public class C4{
       C2 v;
     }
  }
}
```

Program 15.3 *Two namespaces and a nested namespace with classes.*

In Program 15.4 we show how to use the classes `C1`, `C2`, `C3`, and `C4` from Program 15.3. The **using** directives import the types of a namespace. Importing a namespace `N` implies that types `T` in `N` can be used without qualification. Thus, we can write `T` instead of `N.T`. The three **using** directives in line 15-17 of Program 15.4 open up the namespaces `N1`, `N2` and `N2.N3`. If the namespaces in Program 15.3 and the client class shown in Program 15.4 are compiled to two different assemblies (dll files) then `C2` cannot be used in the `Client` class. The reason is that `C2` is internal in its assembly.

```
1  // A client program
2  // In source file client.cs
3
4
5  /*
6    Namespace N1
7      public class C1
8    Namespace N2
9      internal class C2
10     public class C3
11     Namespace N3
12       public class C4
13 */
14
15 using N1;
16 using N2;
17 using N2.N3;
18
19 public class Client{
20   C1 v = new C1();
21
22   // The type or namespace name 'C2' could not be found.
23   // C2 w = new C2();
24   C3 x = new C3();
25   C4 y = new C4();
26 }
```

Program 15.4   *A client of classes in different namespaces.*

If you avoid the **using** directives, you are punished with the need to use a lot of "namespace dotting". If you wish to see the effect of this, please consult Program 15.5 (only on web)

The compilation of Program 15.3 together with Program 15.5 and Program 15.4 (only on web) is shown in Listing 15.6 (only on web).

Nested namespaces can be given by textual nesting, as shown in Program 15.3 or in Program 15.7 (only on web). Alternatively, it can be given as shown in Program 15.8. In Program 15.8 the namespaces `N2` and `N3` are both member of the namespace `N1`. Thus, the situation in Program 15.8 is identical to the situation shown in Program 15.7 (only on web).

```
// Equivalent to the previous program
// No physical namespace nesting
// In source file ex-equiv.cs

namespace N1.N2 {
  public class C1{};
  public class C2{};
}

namespace N1.N3 {
  public class C3{}
}
```

Program 15.8 *Equivalent program with nested namespaces - no physical nesting.*

The classes `C1`, `C2`, and `C3` of either Program 15.8 or Program 15.7 (only on web) can be used in a `Client` class, as shown in Program 15.9 (only on web). The compilation can be done as shown in Listing 15.10 (only on web).

A namespace, such as `Intro` in Program 15.11 is open ended in the sense that stuff can be added to `Intro` from another source file. Both Program 15.11 and Program 15.12 contribute to the `Intro` namespace. Thus, when the two source files are taken together, `Intro` contains the types `A`, `B`, and `C`. The use of the namespace `Intro` is shown in `Client` class in Program 15.13 (only on web). In Listing 15.14 (only on web) we show how to compile the two source files `f1.cs` and `f2.cs` behind the namespace `Intro` together.

```
// f1.cs: First part of the namespace Intro

using System;

namespace Intro{

  internal class A {

    public void MethA (){
      Console.WriteLine("This is MethA in class Intro.A");
    }
  }

  public class B {

    private A var = new A();

    public void MethB (){
      Console.WriteLine("This is MethB in class Intro.B");
      Console.WriteLine("{0}", var);
    }
  }

}
```

Program 15.11 *Part one of namespace Intro with the classes A and B.*

```
// f2.cs: Second part of the namespace Intro

using System;

namespace Intro{

  public class C {
    private A var1 = new A();
    private B var = new B();

    public void MethC (){
      Console.WriteLine("This is MethC in class Intro.C");
      Console.WriteLine("{0}", var);
      Console.WriteLine("{0}", var1);
    }
  }
}
```

Program 15.12 *Part two of namespace Intro with the class C.*

The problem reported in line 18 of Program 15.13 relies on the compilation of the program to two different assemblies, as shown in Listing 15.14. If both the `Intro` namespace and the `Client` class are compiled to a single assembly there will be no error in line 18.

The compilations shown in Listing 15.14 illustrate how to compile the files `f1.cs` and `f2.cs` together. In general, it is possible to compile a number of C# source files together as though these source files were contained in a single large source file. This way of compilation is often an easy way to compile a number of C# source files that depend on each other in circular ways. Alternatively, each file must be compiled in isolation and in a particular order, with use of the `reference` compiler option.

Notice also, from Listing 15.14, that you can control the name of the assembly via use of the `out` compiler option.

15.3. Namespaces and Visibility

Lecture 4 - slide 20

In this section we summarize the visibility rules of types and namespaces, both of which can occur in (other) namespaces.

- Types declared in a namespace
  - Can either have public or internal access
  - The default visibility is internal
  - Internal visibility is relative to an assembly - not a namespace
- Namespaces in namespaces
  - There is no visibility attached to namespaces
  - A namespace is implicitly public within its containing namespace

You should pay attention to the default visibility of types in namespaces. If you do not give a visibility modifier of a type `T` (a class, for instance) in a namespace `N`, `T` is internal in `N`. This may lead to surprises if you in reality forgot to state that `T` should have been public. We have already discussed this in Section 11.16.

15.4. Namespaces and Assemblies

Lecture 4 - slide 21

- Namespaces
  - The top-level construct in a compilation unit
  - May contain types (such as classes) and nested namespaces
  - Identically named members of different namespaces can co-exist
  - There is no coupling between classes/namespaces and source files/directories
- Assemblies
  - A packaging construct produced by the compiler
    - Not a syntactic construct in C#
  - A collection of compiled types - together with resources on which the types depend
  - Versioning and security settings apply to assemblies

The **file/directory** organization, the **namespace/class** organization and the **assembly** organization are relatively independent of each other

15.5. References

[Ecma-334] "The C# Language Specification", June 2005. ECMA-334.

16. Patterns and Techniques

Throughout this material we there will be chapters titled "Patterns and Techniques". A number of such chapters are oriented towards object-oriented design patterns. In Section 16.1 we therefore introduce the general idea of design patterns, and in Section 16.2 we specialize this to a discussion of object-oriented design patterns. In Section 16.3 we encounter the first object-oriented design pattern, the one called ***Singleton***. In Section 16.5 we discuss how to avoid leaking private information from a class.

16.1. Design Patterns

Lecture 4 - slide 23

Design patterns originate from the area of architecture, and they were pioneered by the architect Christopher Alexander.

The following is an attempt to give a very dense and concentrated definition of design patterns.

A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces [Brad Appleton]

Each of the important, underlined words - and a few more - are addressed below:

- *Named:* Eases communication about problems and solutions
- *Nugget:* Emphasizes the value of the pattern
- *Recurring problem:* A pattern is intended to solve a problem that tends to reappear.
- *Proven solution:* The solution must be proven in a number of existing programs
- *Nontrivial solution:* We are not interested in collecting trivial and obvious solutions
- *Context:* The necessary conditions and situation for applying the pattern
- *Forces:* Considerations and circumstances, often in mutual conflict with each other

A set of design patterns serve as a catalogue of well-proven solutions to (more or less) frequently occurring problems. A design pattern has a name that eases the communication among programmers. A design pattern typically reflects a solution to a problem, which is non-trivial and distanced from naive and obvious solutions.

16.2. Object-oriented Design Patterns

Lecture 4 - slide 24

Object-oriented design patterns were introduced in the book "*Design Patterns - Elements of Reusable Object-Oriented Software*" by Gamma, Helm, Johnson and Vlissides.

Numerous books have been written about design patterns (and other kinds of patterns as well). The book mentioned above, [Gamma96], was the first and original one, and it still has a particular status in the area. It is often referred to as the GOF (Gang of Four) book. The patterns and pattern categories mentioned below stem from the original book.

- Twenty three patterns categorized as
    - Creational Patterns
        - Abstract factory, Factory method, Singleton, ...
    - Structural Patterns
        - Adapter, Composite, Proxy, ...
    - Behavioral Patterns
        - Command, Iterator, Observer, ...

There are patterns in a variety of different areas, and at various levels of abstractions

16.3. The Singleton pattern

Lecture 4 - slide 25

The concrete contribution of this chapter is the ***Singleton*** design pattern. As stated below, use of ***Singleton*** is intended to ensure that a given class can be instantiated at most once.

Problem: For some classes we wish to guarantee that at most one instance of the class can be made.

Solution: The singleton design pattern

The idea of ***Singleton*** is to remove the constructor from the client interface. This is done by making it private. Instead of the constructor the class provides a public static method, called `Instance` in Program 16.1, which controls the instantiation of the class. Inside the `Instance` method the private constructor is available, of course. The private, static variable `uniqueInstance` keeps track of an existing instance (if it exists). If there already exists an instance, the `Instance` method returns it. If not, `Instance` creates an instance and assigns it to the variable `uniqueInstance` for future use. All this appears in Program 16.1.

```
public class Singleton{

  // Instance variables

  private static Singleton uniqueInstance = null;

  private Singleton(){
     // Initialization of instance variables
  }

  public static Singleton Instance(){
    if (uniqueInstance == null)
      uniqueInstance = new Singleton();

    return uniqueInstance;
  }

  // Methods

}
```

Program 16.1   *A template of a singleton class.*

Let us program a singleton `Die` class. It is shown below in Program 16.2. We have already seen the `Die` class in Section 10.1 (Program 10.1) and Section 12.4 (Program 12.5).

It should be easy to recognize the pattern from Program 16.1 in Program 16.2.

```
using System;

public class Die {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private int maxNumberOfEyes;

  private static Die uniqueInstance = null;

  private Die (){
    randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
    this.maxNumberOfEyes = 6;
    Toss();
  }

  public static Die Instance(){
    if (uniqueInstance == null)
      uniqueInstance = new Die();

    return uniqueInstance;
  }

  // Die methods: Toss and others

}
```

Program 16.2   *A singleton Die class.*

Let us know bring the singleton `Die` class into action. It is done in Program 16.3. First notice that we cannot just instantiate the singleton `Die` class. The compiler will complain. In Program 16.3 we attempt to make two dice with use of the `Instance` method. In reality, the second call of `Instance` returns the same die as returned by the first call of `Instance`. Thus, `d2` and `d3` refer to the same object. The program first tosses the die referred by `d2` four times, and next it tosses the die referred by `d3` five times. In reality *the same die* is tossed nine times. The output of the die tossing program is shown in Listing 16.4 (only on web).

Recall our very first class example in Section 10.1. In Program 10.2 the three different `Die` objects tossed in identical ways. The reason is that they use three separate - but identically seeded - `Random` objects. The solution is to use a singleton `Random` class, which ensures that at most a single `Random` object can exist. The three `Die` objects will share the `Random` object. With this organization we solve the "parallel tossing problem". Please consult Exercise 3.7 and its solution.

```
using System;

class diceApp {

  public static void Main(){

//  Die d1 = new Die();   //  Compile-time error:
                          //  The type 'Die' has no constructors defined
```

```

    Die d2 = Die.Instance(),
        d3 = Die.Instance();

    for(int i = 1; i < 5; i++){
      Console.WriteLine(d2);
      d2.Toss();
    }

    for(int i = 5; i < 10; i++){
      Console.WriteLine(d2);
      d3.Toss();
    }

    // Test for singleton:
    if (d2 == d3)
      Console.WriteLine("d2 and d3 refer to same die instance");
    else
      Console.WriteLine("d2 and d3 do NOT refer to same die instance");
 }

}
```

Program 16.3   *Application of the singleton Die class.*

You may ask if ***Singleton*** is important in everyday programming. How often do we have a class that only can give rise to one object? The singleton `Die` shown above is not a very realistic use of ***Singleton***.

Singleton is probably not the most frequently used pattern. But every now and then we encounter classes, which it does not make sense to instantiate multiple times. In these situations is it nice to know how to proceed. Use ***Singleton*** instead of a homegrown ad hoc solution! There are additional details which can be brought up in the context of Singleton, see [singleton-msdn].

16.4. Factory methods

Lecture 4 - slide 27

As we have seen in Chapter 12, instantiation of classes by use of programmed constructors is the primary means for creation of new objects. In some situations, however, direct use of constructors is not flexible enough. In this section we will see how we can make good use of static methods as a supplementary means for object creation. Such methods are called *factory methods*.

We have already studied class `Point` several times, see Section 11.6 and Section 14.6. In the version of class `Point` shown in Program 16.5 below we need constructors for both polar and rectangular initialization of points. Recall that rectangular represented points have ordinary *(x,y)* coordinates and that polar represented points have *(r,a)* - radius and angle - coordinates. If we use two constructors for the initialization, both will take two double parameters. In Program 16.5 we supply an extra enumeration parameter to the last constructor, shown in line 16. This is an ugly solution.

```
using System;

public class Point {

  public enum PointRepresentation {Polar, Rectangular}
  private double r, a;    // polar data repr: radius, angle

  // Construct a point with polar coordinates
  public Point(double r, double a){
     this.r = r;
     this.a = a;
  }

  // Construct a point, the representation of which depends
  // on the third parameter.
  public Point(double par1, double par2, PointRepresentation pr){
   if (pr == PointRepresentation.Polar){
     r = par1; a = par2;
   }
   else {
     r = RadiusGivenXy(par1,par2);
     a = AngleGivenXy(par1,par2);
   }
  }

  private static double RadiusGivenXy(double x, double y){
    return Math.Sqrt(x * x + y * y);
  }

  private static double AngleGivenXy(double x, double y){
    return Math.Atan2(y,x);
  }

  // Remaining Point operations not shown
}
```

Program 16.5 *A clumsy attempt with two overloaded constructors.*

In Program 16.6 we show another version, in which the constructor is private. From the outside, the two static factory methods `MakePolarPoint` and `MakeRectangularPoint` are used for construction of points. Internally, these methods delegate their work to the private constructor. This is a much more symmetric solution than Program 16.5, and it allows us to have good names for the "constructors" - or more correctly, the *factory methods*.

```
using System;

public class Point {

  public enum PointRepresentation {Polar, Rectangular}

  private double r, a;     // polar data repr: radius, angle

  // Construct a point with polar coordinates
  private Point(double r, double a){
     this.r = r;
     this.a = a;
  }

  public static Point MakePolarPoint(double r, double a){
    return new Point(r,a);
  }

  public static Point MakeRectangularPoint(double x, double y){
    return new Point(RadiusGivenXy(x,y),AngleGivenXy(x,y));
  }

  private static double RadiusGivenXy(double x, double y){
    return Math.Sqrt(x * x + y * y);
  }

  private static double AngleGivenXy(double x, double y){
    return Math.Atan2(y,x);
  }

  // Remaining Point operations not shown
}
```

Program 16.6 *A better solution with static factory methods.*

In the web-edition of this material we present another example of factory methods. This example is given in the context of the `Interval` struct, which we will encounter in Section 21.3. The constructor problem of this type is that structs do not allow parameterless constructors. It is, however, natural for us to have a parameterless constructor for an empty interval. Program 16.7 (only on web) shows a clumsy solution, and Program 16.8 (only on web) shows a more satisfactory solution that uses a factory method.

In Section 32.10 we come back to factory methods, and in particular to an object-oriented design pattern called ***Factory Method***, which relies on inheritance.

Chose a coding style in which *factory methods* are consistently named: **`Make...(...)`**

16.5. Privacy Leaks

Lecture 4 - slide 29

The discussion in this section is inspired by the book *Absolute Java* by Walter Savitch. Privacy leaks is normally not thought of as a design pattern.

Problem: A method can return part of its private state, which can be mutated outside the object

To be concrete, let us look at the problem in context of Program 16.9 and Program 16.10. We use properties in this example. Properties will be introduced in Chapter 18. On the slide belonging to this example we show a version with methods instead. The class `Person` represents the birth date as a `Date` object. In order to make our points clear we provide a simple implementation of the `Date` class in Program 16.9. In real-life programming we would, of course, use C#'s existing `DateTime` struct. You should notice that the property `DateOfBirth` in line 17-19 of Program 16.10 returns a reference to a private `Date` object, which represents the person's birthday.

The client of class `Person`, shown in Program 16.11 mutates the `Date` object referred by `d`. The mutation of the `Date` objects takes place in line 10. This object came from the birthday of person p. Is this at all reasonable to do so, you may ask. I would answer "yes". If you have access to a mutable `Date` object chances are that you will forget were it came from, and eventually you may be tempted to modify (mutate) it.

As shown in the output of the client program, in Listing 16.12, Hanne is now 180 years old. We have managed to modify her age despite the fact the birthday is private in class `Person`.

As of now we leave it as an exercise to find good solutions to this problem, see Exercise 4.3.

```
public class Date{
  private ushort year;
  private byte month, day;

  public Date(ushort year, byte month, byte day){
    this.year = year; this.month = month; this.day = day;
  }

  public ushort Year{
    get{return year;}
    set{year = value;}
  }

  public byte Month{
    get{return month;}
    set{month = value;}
  }

  public byte Day{
    get{return day;}
    set{day = value;}
  }

  public override string ToString(){
    return string.Format("{0}.{1}.{2}",day, month, year);
  }
}
```

Program 16.9 *A Mutable Date class.*

```
public class Person{

  private string name;
  private Date dateOfBirth, dateOfDeath;

  public Person (string name, Date dateOfBirth){
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    this.dateOfDeath = null;
  }

  public string Name{
    get {return name;}
    set {name = value;}
  }

  public Date DateOfBirth{
    get {return dateOfBirth;}
  }

  public ushort AgeAsOf(Date d){
    return (ushort)(d.Year - dateOfBirth.Year);
  }

  public bool Alive(){
    return dateOfDeath == null;
  }

  public override string ToString(){
    return "Person: " + name + " " + dateOfBirth;
  }

}
```

Program 16.10 *A Person class that can return its private birth Date.*

```
using System;

class Client{

  public static void Main(){

    Person p = new Person("Hanne", new Date(1926, 12, 24));

    Date d = p.DateOfBirth;
    d.Year -= 100;
    Console.WriteLine("{0}", p);

    Date today = new Date(2006,8,31);
    Console.WriteLine("Age of Hanne as of {0}: {1}.",
                      today, p.AgeAsOf(today));
  }

}
```

Program 16.11 *A client of the Person which modifies the returned birth Date.*

```
Person: Hanne 24.12.1826
Age of Hanne as of 31.8.2006: 180.
```

Listing 16.12 *The output of the Person client program.*

Exercise 4.3. *Privacy Leaks*

The starting point of this exercise is the observations about *privacy leaks* on the accompanying slide.

Make sure that you understand the problem. Test-drive the program (together with its dependent `Person` class and `Date` class) on your own computer.

If you have not already done so, read the section about privacy leaks in the textbook!

Find a good solution to the problem, program it, and test your solution.

Discuss to which degree *you* will expect that this problem to occur in everyday programming situations.

We return to the `Date` and `Person` classes in Section 20.4 and Section 20.5. In these sections we also comment on the privacy leak problem.

16.6. References

[Gamma96]	E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, 1996.
[Singleton-msdn]	MSDN: Implementing Singleton in C# http://msdn.microsoft.com/en-us/library/ms998558.aspx