10. Classes: An Initial Example

This is the first chapter about classes. It is also the first chapter in the first lecture about classes. Our basic coverage of classes runs until Chapter 13.

10.1. The Die Class

Lecture 3 - slide 2

In this section we encounter a number of important OOP ideas, observations, and principles. We will very briefly preview many of these in a concrete way in the context of a simple initial class. Later we will discuss the ideas in depth.

We use the example of a *die*, which is the singular form of "dice", see Program 10.1. One of the teaching assistants in 2006 argued that the class `Die` is a sad beginning of the story about classes. Well, it is maybe right. I think, however, that the concept of a die is a good initial example. So we will go for it!

On purpose, we are concerned with use of either the singular or the plural forms of class names. The singular form is used when we wish to describe and program a single phenomenon/thing/object. The plural form is most often used for collections, to which we can add or delete (singular) objects. Notice that we can make multiple instances of a class, such as the `Die` class. In this way we can create a number of dice.

The class `Die` in Program 10.1 is programmed in C#. We program a die such that each given die has a fixed maximum number of eyes, determined by the constant `maxNumberOfEyes`. The class *encapsulates* the *instance variables*: `numberOfEyes`, `randomNumberSupplier`, and the constant `maxNumberOfEyes`. They are shown in line 4-6. The instance variable are intended to describe the *state* of a `Die` object, which is an *instance* of the `Die` class. The instance variable `numberOfEyes` is the most important variable. The variable `randomNumberSupplier` makes it possible for a `Die` to request a random number from a *Random* object.

After the instance variables comes a constructor. This is line 8-11. The purpose of the constructor is to initialize a newly create `Die` object. The constructor makes the random number supplier, which is an instance of the `System.Random` class. The constructor happens to initialize a normal six-eyed die. The expression `DateTime.Now.Ticks` returns a `long` integer, which we type cast to an `int`. The use of an `unchecked` context implies that we get an `int` out of the cast, even if the `long` value does not fit the range of `int`. (The use of `unchecked` eliminates overflow checking). The value assigned to `numberOfEyes` is achieved by tossing the die once via activation of the method `NewTossHowManyEyes`. The call of `NewTossHowManyEyes` on line 10 delivers a number between 1 and 6. In this way, the initial state - the number of eyes - of a new die is random.

Then follows three operations. In most object-oriented programming languages the operations are called *methods*. The `Toss` operation modifies the value of the `numberOfEyes` variable, hereby simulating the tossing of a die. The `Toss` operation makes use of a private method called `NewTossHowManyEyes`, which interacts with the random number supplier. The `NumberOfEyes` method just accesses the value of the instance variable `numberOfEyes`. The `ToString` method delivers a string, which for instance can by used if we decide "to print a Die object". The `ToString` method in class `Die` overrides a more general method of the same name.

We notice that the instance variables are private and that the constructors and methods are public. Private instance variables cannot be used/seen from other classes. This turns out to be important for us, see Section 11.4.

```
using System;

public class Die {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private const int maxNumberOfEyes = 6;

  public Die(){
    randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
    numberOfEyes = NewTossHowManyEyes();
  }

  public void Toss(){
    numberOfEyes = NewTossHowManyEyes();
  }

  private int NewTossHowManyEyes (){
    return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
  }

  public int NumberOfEyes() {
    return numberOfEyes;
  }

  public override String ToString(){
    return String.Format("[{0}]", numberOfEyes);
  }
}
```

Program 10.1 *The class Die.*

Below, in Program 10.2 we see a client of class `Die`, which creates and repeatedly tosses three dice. A client of a `Die` uses a die via a number of `Die` references. In Program 10.2 `d1`, `d2`, and `d3` are references to `Die` objects. Section 10.2 is about clients and servers.

When we run the program we get the output shown in Listing 10.3

```
using System;

class diceApp {

  public static void Main(){

    Die d1 = new Die(),
        d2 = new Die(),
        d3 = new Die();

    for(int i = 1; i < 10; i++){
      Console.WriteLine("Die 1: {0}", d1);  // Implicitly
      Console.WriteLine("Die 2: {0}", d2);  // calls
      Console.WriteLine("Die 3: {0}", d3);  // ToString in Die
      d1.Toss();  d2.Toss();  d3.Toss();
    }

 }
}
```

Program 10.2 *A program that tosses three dice.*

```
Die 1: [1]
Die 2: [1]
Die 3: [1]
Die 1: [2]
```

```
Die 2: [2]
Die 3: [2]
Die 1: [3]
Die 2: [3]
Die 3: [3]
Die 1: [4]
Die 2: [4]
Die 3: [4]
Die 1: [3]
Die 2: [3]
Die 3: [3]
Die 1: [2]
Die 2: [2]
Die 3: [2]
Die 1: [3]
Die 2: [3]
Die 3: [3]
Die 1: [2]
Die 2: [2]
Die 3: [2]
Die 1: [1]
Die 2: [1]
Die 3: [1]
```

Listing 10.3 *Sample program output.*

The output shown in Program 10.1 seems suspect. Why? Take a close look. We will come back to this problem in Exercise 3.7, which we encounter in Section 11.10. (At that location in the material we have learned enough to come up with a good solution to the problem).

> The Die class is a *template* or *blueprint* from which we can create an arbitrary number of die objects

The term blueprint is often used as a metaphor of a class (seen in relation to objects). The word 'blueprint' is, for instance, used for an architect's drawing of a house. In general, a blueprint refers to a detailed plan for something that, eventually, is going to be constructed. The blueprint can be used as a prescription from which craftsmen can actually built a house.

The class `Die` from Program 10.1 is only useful if we apply it in some context where dice are actually needed. We use dice in various games. In Exercise 3.1 we propose that you make a simple Yahtzee game, with use of five dice (five instances of the `Die` class).

Exercise 3.1. *Yahtzee*

Write a very simple Yahtzee program based on the `Die` class. Yahtzee is played by use of five dice that are tossed simultaneously. The players are supposed to fill out a table of results. The table should allow registration of ones, ..., sixes, three-of-a-kind, four-of-a-kind, full-house, small-straight, large-straight, yahtzee, and chance. See wikipedia for more details and inspiration.

Be sure to use the version of the `Die` class that shares the `Random` class with the other dice. This version of the `Die` class is produced in another exercise in this lecture.

This program is for a single player, who tosses the five dice repeatedly. Each time the five dice are tossed a table cell is filled. No re-tossing is done with less than five dice. The first category that fits a given toss is

filled. (Try yahtzee first, Chance last). Keep it simple!

You may consider to write a `YahtzeeTable` class which represents the single user table used to register the state of the game. Consider the interface and operations of this class.

10.2. Clients and Servers

Lecture 3 - slide 3

The names *client* and *server* is often used when we are concerned with computers. A server denominate a computer that provides services to its surrounding. It could be a file server or a web server.

In the context of object-oriented programming the words client and server will be used as *object roles*. In a given situation an object plays a given role. An object **`x`** is called a client of **`y`** if **`x`** makes use of the services (operations) provided by **`y`**. An object **`y`** is called a server of **`x`** if it provides services (operations) to **`x`**.

In a dice game the objects act, in turn, as *client* and *servers*.

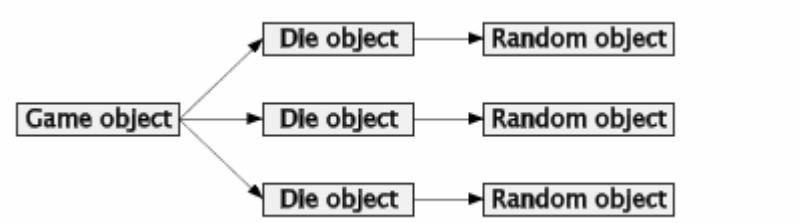

Figure 10.1 *Interacting Game, Die, and Random objects. The Game object is a client of the Die objects, which in turn are clients of the Random objects.*

Figure 10.1 shows a single game object, three die objects, and three random objects. The client-server roles of these objects can be summarized as follows:

- The **`Game`** object is a client of a number of **`Die`** objects
- A given **`Die`** object is a client of a **`Random`** object
- In turn, a **`Die`** object act as a server for the **`Game`** object, and **`Random`** objects act as servers for **`Die`** objects

In the figure, the arrows are oriented from clients to servers.

10.3. Message Passing

Lecture 3 - slide 4

A client interacts with its connected servers via message passing.

As a metaphor, we pretend that objects communicate by means of *message passing*

Message passing actually covers a procedure call. Procedure calling is a technical matter. Message passing is an everyday term that covers some communication between one person and another, for instance via postal mail. In some setups, message passing also involves the receiving of a reply. As already stressed earlier, use of metaphors is very important for getting new ideas, and for raising the level of abstraction.

In Figure 10.2 we illustrate message passing in between a game object, three dice, and three random objects.

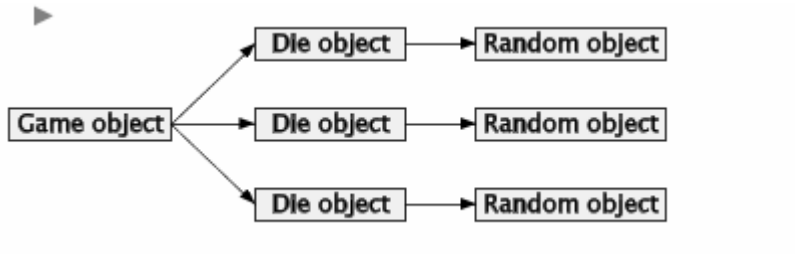

Figure 10.2 *Interacting* ***`Game`***, ***`Die`***, *and* ***`Random`*** *objects*

In some versions of this teaching material you will be able to animate the figure, such that you can actually see the acts of passing a message and receiving a reply. Before sending a message, the sending object is emphasized. When emphasized, we say that the object is the *current object*. In a single threaded program execution, there is always a single current object. This is the object, which most recently received a message. In football, message passing corresponds to passing the ball from player to player. At some level of abstraction, there is always a single player - 'the ball keeper' - who posses the ball. He or she corresponds to the current object.

Here follows some general remarks about message passing.

- We often prefer to think of the interaction between objects as message passing.
- The receiver of an object locates a procedure or a function which can answer the message - *method lookup*
- A result may be sent back from the receiver to the sender of the message.

In the next chapter we will dive into the details of the class concept.

11. Classes

The most important programming concept in object-oriented programming is the class. The programmer writes the classes in his or her source program. At run time, classes are used as blueprints/templates for instantiation of classes (creation of objects). In this chapter we will explore the concept of classes. This will be a relatively long journey through visibility issues, representation independence, instance and class variables, instance and class methods, and the notation of the current object. At the end of the chapter, in Section 11.14 we will discuss the important differences between classes and objects.

11.1. Classes

Lecture 3 - slide 6

The single most important aspect of classes is *encapsulation*. As a matter of fact, I believe that the most important achievement of object-oriented programming is the idea of *systematic encapsulation of variables and operations that belong together*.

A class is a construct that surrounds a number of definitions, which belong together. Some of these definitions can be seen from the outside, whereas others are only relevant seen from the inside. Here follows a short 'definition' of a class:

> A *class* encapsulates data and operations that belong together, and it controls the visibility of both data and operations. A class can be used as a type in the programming language

The parts of a class which are visible from other classes forms the *client interface* of the class. In the figure, the interface of a class is drawn on the border of the box that surrounds the variables and operations. Thus, in the figure, only a subset of the operations - Op1, Op2, Op3, and Op4 - form the client interface of the class. All data parts are kept inside the class, and they cannot be directly used from other classes.

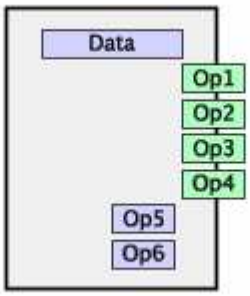

Figure 11.1 *A class and its interface to other classes. The interface is often called the client interface. In this illustration the operations Op1, Op2, Op3, and Op4 form the client interface of the class.*

The notion of interfaces between program parts - program building blocks - is important in general. In this section we talk about the interfaces between classes. It turns out that a class may have several different interfaces. The interface we care most about right now is called the *client interface* of a class C. There is another interface between C and classes that extends or specializes C. We have more to say about this interface in Section 27.2.

Exercise 3.2. *Time Classes*

This is not a programming exercise, but an exercise which asks you to consider data and operations of classes related to time.

Time is very important in our everyday life. Therefore, many of the programs we write somehow deal with time.

Design a class `PointInTime`, which represents a single point in time. How do we represent a point in time? Which variables (data) should be encapsulated in the class? Design the variables in terms of their names and types.

Which operations should constitute the client interface of the class? Design the operations in terms of their names, formal parameters (and their types), and the types of their return values.

Can you imagine other time-related classes than `PointInTime`?

Avoid looking at the time-related types in the C# library before you solve this exercise. During the course we will come back the time related types in C#.

11.2. Perspectives on classes

Lecture 3 - slide 7

In this section we discuss different ways to understand classes relative to already established understandings. You may safely skip this section if such discussion does not appeal to you.

Depending on background and preferences, different programmers may have different understandings of classes. Here follows some of these.

- Different perspectives on classes:
  - An abstract datatype
  - A generalization of a record (struct)
  - A definition procedure
  - A module

Types and *abstract datatypes* are topics of general importance in computer science. But it is probably fair to state that the topic of types is of particular importance in the theoretical camp. Abstract datatypes have been studied extensively by mathematically inclined computer scientists, not least from an interest of specification. Boiled down to essence, a type can be seen as a set of values that possess a number of common properties. An abstract datatype is a set of such values and a set of operations on these values. The operations make the values useful. When we talk about abstract data types, the data details of the values in the type are put behind the scene.

In most imperative programming language, including Pascal and C, a record (a struct in C) is a data structure that groups data together. We often say that data parts are *aggregated* in a record. Records are called structs in C. It is a natural and nice idea to organize the operations of the grouped data together with the data

themselves; In other words, to 'invite' the operations on records/structs into the record itself. In C#, structs are used as a "value variant" of a class. This is the topic in Section 14.1.

Abstractions can be formed on top of expressions. This leads to the functions. In the same way, procedures are abstractions of commands/statements. A call of a function is itself an expression, and a call of a procedure is a command/statement. From a theoretical point of view it is possible to abstract other syntactic categories as well, including a set of definitions. Such abstractions have been called *definition procedures* [Tennent81]. Classes can therefore be seen as definition procedures. Following the pattern from above, the activation of a definition procedure leads to definitions. It is not obvious, however, if multiple activations of a definition procedure is useful.

Finally, a module is an encapsulation, which does not act as a type. A module may, on the other hand, contain a type (typically a struct) that we treat as an abstract datatype. See Section 2.3 for our earlier discussion of modules.

11.3. Visibility - the Iceberg Analogy

Lecture 3 - slide 8

As stated in Section 11.1 visibility-control is an important aspect of classes. Inspired by Bertrand Meyers seminal book *Object-oriented software construction*, [Meyer88], we will compare a class with an iceberg.

A class can be seen as an *iceberg*: Only a minor part of it should be visible from the outside. The majority of the class details should be hidden.

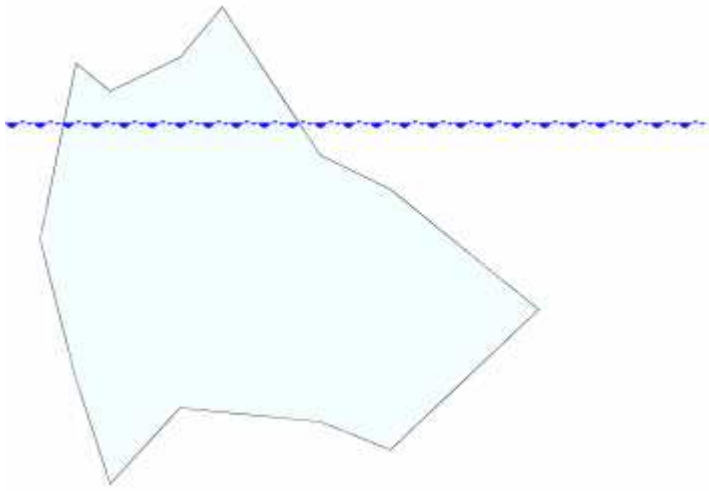

Figure 11.2 *An Iceberg. Only a minor fraction of the iceberg is visible above water. In the same way, only a small part of the details of a class should be visible from other classes.*

Clients of a class C cannot directly depend on hidden parts of C.

Thus, the invisible parts in C can more easily be changed than the parts which constitute the interface of the class.

Visibility-control is important because it protects the invisible parts of a class from being directly accessed from other classes. No other parts of the program can rely directly on details, which they cannot access. If some detail (typically a variable) of a class cannot be seen outside the class, it is much easier to modify this detail (e.g. replace the variable by a set of other variables) at a later point in time.

You may ask why we would like to modify details of our class. We can, of course, hope that we do not need to. But if the program is successful, and if it is alive many years ahead, it is most likely that we need to change it eventually. Typically, we will have to extend it somehow. It is also typical that we have to change the representation of some of our data. It is very costly if these changes cause a ripple effect that calls for *many* modifications throughout the whole program. It is very attractive if we can limit the area of the program that needs attention due to the modification. A programmer who use a programming language that guaranties a given visibility control policy is in a good position to deal with the consequences of the mentioned program modifications.

11.4. Visible and Hidden aspects

Lecture 3 - slide 9

Let us now be more concrete about class visibility. In this section we will describe which aspect to keep as class secrets, and which aspect to spread outside the class.

- Visible aspects
  - The name of the class
  - The signatures of selected operations: The interface of the class
- Hidden aspects
  - The representation of data
  - The bodies of operations
  - Operations that solely serve as helpers of other operations

The visible aspects should be kept at minimum level. The class name must be visible. The major interface of the class is formed by the signatures of selected operations. A *signature of a method* is the name of the method together with the types of the method parameters, and the type of the value returned by the method.

It is always recommended to keep the representation of data secret. It is almost always wrong to export knowledge about the instance variables of a class. Clients of the class should not care about - and should not know - data details. If we reveal data details it is very hard to change the data presentation at a later point in time. Let us stress again that it is a very typical modification of a program to alter the representation of data.

The bodies of the operations (the operation details beyond the operation signature) are hidden because operations are themselves abstractions (of either expressions or command). Finally, some operations serve as helper operations in order to encourage internal reuse within the class, and in order to prevent the operations of the class to become too large. Such helper operations should also be invisible to the clients of the class.

In Program 11.1 we show and emphasize the visible parts of the `Die` class from Program 10.1. We have dimmed the aspects of the `Die` class which are invisible to client classes (the aspects 'below the surface' relative to Figure 11.2).

```
1  using System;
2
3  public class Die {
4    private int numberOfEyes;
5    private Random randomNumberSupplier;
6    private const int maxNumberOfEyes = 6;
7
8    public Die(){
9      randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
10     numberOfEyes = NewTossHowManyEyes();
11   }
12
13   public void Toss(){
14     numberOfEyes = NewTossHowManyEyes();
15   }
16
17   private int NewTossHowManyEyes (){
18     return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
19   }
20
21   public int NumberOfEyes() {
22     return numberOfEyes;
23   }
24
25   public override String ToString(){
26     return String.Format("[{0}]", numberOfEyes);
27   }
28 }
```

Program 11.1 *The class Die - aspects visible to clients emphasized.*

Some programming language enforce that all instance variables of a class are hidden. Smalltalk [Goldberg83] is one such language. C# is not in this category, but we will typically strive for such discipline in the way we program in C#.

11.5. Program modification - the Fire Analogy

Lecture 3 - slide 10

In continuation of the iceberg analogy, which illustrated visibility issues, we will here illustrate program modification issues by an analogy to the spread of fire.

> A minor modification of a program may spread as a fire throughout the program.
>
> Such a fire may ruin most of the program in the sense that major parts of the program may need to be reprogrammed.

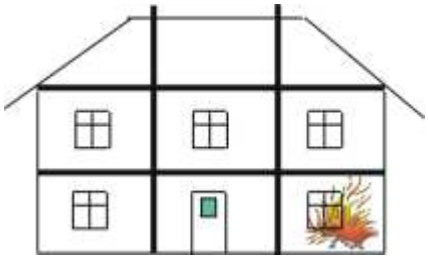

Figure 11.3 *A house - with firewalls - on fire. The fire is not likely to spread to other apartments because of the solid firewalls.*

The use of firewalls prevents the spread of a fire.

Similarly, encapsulation and visibility control prevent program modifications from having global consequences.

In large buildings, firewalls prevent a fire to destroy more than a single part of a building. Similarly, fire roads in forest areas are intended to keep fires to localized regions of the forest.

11.6. Representation Independence

Lecture 3 - slide 11

Let us now coin an important OOP programming principle: The principle of representation independence.

Representation independence: Clients of the class C should not be affected by changes of C's data representation

In essence, this is the idea we have already discussed in Section 11.4 and Section 11.5. Now we have name for it!

Below, in Program 11.2 we will show a class that is vulnerable in relation to the principle of representation independence. The class is written in C#. The class `Point` in Program 11.2 reveals its data representation to clients. This is because `x` and `y` are public. In Program 11.2 `x` and `y` are parts of the client interface of class `Point`.

```
1  // A very simple point with public data representation.
2  // NOT RECOMMENDED because of public data representation.
3
4  using System;
5
6  public class Point {
7    public double x, y;
8
9    public Point(double x, double y){
10    this.x = x; this.y = y;
11   }
12
13   public void Move(double dx, double dy){
14     x += dx; y += dy;
15   }
16
17   public override string ToString(){
```

```
18     return  "(" + x + "," + y + ")";
19   }
20 }
```

Program 11.2 *A Point class with public instance variables - NOT Recommended .*

The class shown below in Program 11.3 is a client of `Point`. It prompts the user for three points that we will assume form the shape of a triangle. In line 24-31 we calculate the circumference of this triangle. In these calculations we use the `x` and `y` coordinates of points directly, and quite heavily!

```
1  // A client of Point that instantiates three points and calculates
2  // the circumference of the implied triangle.
3
4  using System;
5
6  public class Application{
7
8    public static Point PromptPoint(string prompt){
9      double x, y;
10     Console.WriteLine(prompt);
11     x = double.Parse(Console.ReadLine());
12     y = double.Parse(Console.ReadLine());
13     return new Point(x,y);
14   }
15
16   public static void Main(){
17     Point p1, p2, p3;
18     double p1p2Dist, p2p3Dist,  p3p1Dist, circumference;
19
20     p1 = PromptPoint("Enter first point");
21     p2 = PromptPoint("Enter second point");
22     p3 = PromptPoint("Enter third point");
23
24     p1p2Dist = Math.Sqrt((p1.x - p2.x) * (p1.x - p2.x) +
25                          (p1.y - p2.y) * (p1.y - p2.y));
26     p2p3Dist = Math.Sqrt((p2.x - p3.x) * (p2.x - p3.x) +
27                          (p2.y - p3.y) * (p2.y - p3.y));
28     p3p1Dist = Math.Sqrt((p3.x - p1.x) * (p3.x - p1.x) +
29                          (p3.y - p1.y) * (p3.y - p1.y));
30
31     circumference = p1p2Dist + p2p3Dist + p3p1Dist;
32
33     Console.WriteLine("Circumference: {0} {1} {2}: {3}",
34                       p1, p2, p3, circumference);
35
36     Console.ReadLine();
37   }
38
39 }
```

Program 11.3 *A Client of Point.*

Now assume that the programmer of class `Point` changes his or her mind with respect to the representation of points. Instead of using rectangular x and y coordinates the programmer shifts to polar coordinates. This is a representation of points that uses an angle between 0 and 2 pi, and a radius. The motivation behind the shift of representation may easily be that some other programmers request a rotation operation of the class `Point`. It is easy to rotate a "polar point". This leads to a new version of class `Point`, as sketched in Program 11.4. We are, of course, interested in the survival of Program 11.3 and other similar program. Imagine if there exists thousands of similar code lines in other classes!.

```
// A very simple class point with public data representation.
// An incomplete sketch.
// This version uses polar representation.
// NOT RECOMMENDED because of public data representation.

using System;

public class Point {
  public double radius, angle;

  public Point(double x, double y){
    radius = ...
    angle = ...
  }

  public void Move(double dx, double dy){
    radius = ...
    angle = ...
  }

  public void Rotate(double angle){
    this.angle += angle;
  }

  public override string ToString(){
    ...
  }
}
```

Program 11.4 *A version of class Point modified to use polar coordinates - NOT Recommended.*

We will not solve the rest of the problem at this point in time. We leave the solution as challenge to you in Exercise 3.3. In the lecture, which I give based on these notes, I am likely discuss additional elements of a good solution in C#.

Encapsulated data should always be *hidden* and *private* within the class

---

Exercise 3.3. *Public data representation*

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 the accompanying `Point` and `Point` client classes the data representation of `Point` is available to the client class. This may be tempting for the programmer, because we most likely wish to make the x and y coordinates of points available to clients.

Why is it a bad solution? It is very important that you can express and explain the problem to fellow programmers. Give it a try!

Now assume that we are forced (by the boss) to change the data representation of `Point`. As a realistic scenario, we may introduce polar coordinates instead of the rectangular x and y coordinates. Recall that polar coordinates consist of a radius and an angle (in radians or degrees).

What will happen to client classes, such as this client, when this change is introduced? Is it an easy or a difficult modification to the given client class? Imagine that in a real-life situation we can have thousands

of similar lines of code in client programs that refer to x and y coordinates.

Rewrite selected parts of class `Point` such that the client "survives" the change of data representation. In your solution, the instance variables should be private in the `Point` class. Are you able to make a solution such that the client class should not be changed at all?

In the web edition we link to special version of class `Point`, which contains method for conversions between rectangular and polar coordinates. We anticipate that these methods are useful for you when you solve this exercise.

The client class of `Point` calculates the distances between pairs of points. This is not a good idea because far too many details occur repeatedly in the client. Suggest a reorganization and implement it.

11.7. Classes in C#

Lecture 3 - slide 12

In this and the following sections we will study classes in C#, instance variables, instance methods, class variables (static variables), and class methods (static methods).

The syntactic composition of classes is as follows.

```
class-modifiers class class-name{
  variable-declarations
  constructor-declarations
  method-declarations
}
```

Syntax 11.1 *The syntactic composition of a C# Class. This is not the whole story. There are other members than variables, constructors and methods. Notice also that it is NOT required that variables come before constructors and that constructors come before methods.*

Notice, however, that the full story is somewhat more complicated. Inheritance is not taken into account, and only a few class members are listed. In addition, the order of the class members is not constrained as suggested by Syntax 11.1.

The default visibility of members in a class is private. It means that if you do not provide a visibility modifier of a variable or a method, the variable or method will be private. This is unfortunate, because a missing visibility modifier typically signals that the programmer forgot to decide the visibility of the member. It would have been better design of C# to get a compilation error or - at least - a warning.

The following gives an overview of different kinds of members - variables and methods - in a class:

- **Instance variable**
  - Defines state that is related to each individual object
- **Class variable**
  - Defines state that is shared between all objects
- **Instance method**
  - Activated on an object. Can access both instance and class variables
- **Class method**

- Accessed via the class. Can only access class variables

In the following four sections - from Section 11.8 to Section 11.11 - we will study instance variables, instance methods, class variables, and class methods in additional details. This is long journey! You will be back on track in Section 11.12.

11.8. Instance Variables

Lecture 3 - slide 14

All objects of a particular class have the same set of variables. Each object allocates enough memory space to hold its own set of variables. Thus, the values of these variables may vary from one instance (object) to another. Therefore the variables are known as *instance variables*.

> An *instance variable* defines a piece of data in the class. Each object, created as an instance of the class, holds a separate copy of the instance variables.

Unfortunately, the terminology varies a lot. Instance variables are officially known as *fields* in C#. Instance variables are, together with constants, known as *data members*. The term *member* is often used for all declarations contained in a class; This covers data members and function members (constructors, methods, properties, indexers, overloaded operators, and others). Some object-oriented programming languages (Eiffel, for instance) talk about *attributes* instead of instance variables. (In C#, attributes refer to an entirely different concept, see Section 39.6).

Below, in Program 11.5, we show an outline of a `BankAccount` class programmed in C#. The methods are not shown in this version of the class. The class has three instance variables, namely `interestRate` (of type `double`), `owner` (of type `string`), and `balance` (of type `decimal`, a type often used to hold monetary data). In addition the class has three constructors and a number methods, which are not shown here.

```
1  using System;
2
3  public class BankAccount {
4
5     private double interestRate;
6     private string owner;
7     private decimal balance;
8
9     public BankAccount(string owner) {
10       this.interestRate = 0.0;
11       this.owner = owner;
12       this.balance = 0.0M;
13    }
14
15
16    public BankAccount(string owner, double interestRate) {
17       this.interestRate = interestRate;
18       this.owner = owner;
19       this.balance = 0.0M;
20    }
21
22    public BankAccount(string owner, double interestRate,
23                       decimal balance) {
```

```
24        this.interestRate = interestRate;
25        this.owner = owner;
26        this.balance = balance;
27     }
28
29
30     // Remaining methods are not shown here
31  }
```

Program 11.5 *Instance variables in a sketch of the class BankAccount.*

In the `BankAccountClient` class in Program 11.6 we create three different `BankAccount` objects. The variables `a1`, `a2`, and `a3` hold references to these objects.

```
1  using System;
2
3  public class BankAccountClient {
4
5    public static void Main(){
6      BankAccount a1 = new BankAccount("Kurt", 0.02),
7                   a2 = new BankAccount("Bent", 0.03),
8                   a3 = new BankAccount("Thomas", 0.02);
9
10     a1.Deposit(100.0M);
11     a2.Deposit(1000.0M); a2.AddInterests();
12     a3.Deposit(3000.0M); a3.AddInterests();
13
14     Console.WriteLine(a1);   // 100 kr.
15     Console.WriteLine(a2);   // 1030 kr.
16     Console.WriteLine(a3);   // 3060 kr.
17   }
18
19 }
```

Program 11.6 *Creation of three bank accounts.*

Following the calls of the `Deposit` and `AddInterests` operations the three objects can be depicted as shown in Figure 11.4. Please make sure that understand states of the object (the values of the individual instance variables of each of the objects). The output of the program is shown in Figure 11.4. Listing 11.7 (only on web).

Figure 11.4 *Three objects of class BankAccount, each holding three instance variables `interestRate`, `owner`, and `balance`. The values of variables are determined by the bank account transactions that we programmed in the class `BankAccountClient`. The state of the variables is shown relative to the three `WriteLine` calls.*

Exercise 3.4. *How private are private instance variables?*

The purpose of this exercise is to find out how private *private instance variables* are in C#.

Given the `BankAccount` class. Now modify this class such that each bank account has a backup account. For the backup account you will need a new (private) instance variable of type `BankAccount`. Modify the `Withdraw` method, such that if there is not enough money available in the current account, then withdraw the money from the backup account. **As an experiment, access the balance of the backup account directly, in the following way:**

```
backupAccount.balance -= ...
```

Is it possible to modify the private state of one `BankAccount` from another `BankAccount`? Discuss and explain your findings. Are you surprised?

11.9. Instance Methods

Lecture 3 - slide 15

Instance methods are intended to work on (do computations on) the instance variables of an object in a class. An instance method `M` must always be activated on an instance (an object) of the class to which `M` belongs.

Activating or calling an instance method is often thought of as message passing (see Section 2.1). The object, on which the method is activated, is called the receiver of the message. The callee (the object from which the message is sent) is - quite naturally - called the sender.

An *instance method* is an operation in a class that can read and/or modify one or more instance variables.

- An instance method **`M`** in a class **`C`**
  - must be activated on an object which is an instance of **`C`**
  - is activated by ***object*`.M(...)`** from outside **`C`**
  - is activated by **`this.M(...)`** or just **`M(...)`** inside **`C`**
  - can access all members of **`C`**

Notice that an instance method can access all instance variables of a class, including the private ones. An instance method can also access class variables (see Section 11.10).

The form **`object.M(...)`** must be used if a method `M` is activated on an object different from the current object. The short form **`M(...)`** can be used in case `M` is activated on the current object. It is, however, often more clear to write **`this.M(...)`** With this notation we are explicit about the receiver of the message; Also, with the notation **`this.M(...)`**, we use dot notation consistently whenever we activate an instance method. The choice between **`M(...)`** and **`this.M(...)`** depends on the chosen *coding style*. For more details on **`this`** see Section 11.15.

Conceptually you may imagine that each individual object has its own instance methods, in the same way as we in Section 11.8 argued that each individual object has its own instance variables. In reality, however, all instances of a given class can share the instance methods.

Program 11.8 shows a version of the `BankAccount` class in which the instance methods are highlighted. The method `LogTransaction` relies on the enumeration type `AccountTransaction` defined just before the class itself.

In the web-version of the material we show a version of class `BankAccount` with a new instance method `LogTransaction`. This method is used as the starting point of Exercise 3.5.

Exercise 3.5. *The method LogTransaction in class BankAccount*

In the accompanying `BankAccount` class we have sketched and used a private method named `LogTransaction`. Implement this private method and test it with the BankAccount client class.

Exercise 3.6. *Course and Project classes*

In this exercise you are asked to program three simple classes which keep track of the grading of a sample student. The classes are called `BooleanCourse`, `GradedCourse`, and `Project`.

A `BooleanCourse` encapsulates a course name and a registration of passed/not passed for our sample student.

A `GradedCourse` encapsulates a course name and the grade of the student. For grading we use the Danish 7-step, numerical grades 12, 10, 7, 4, 2, 0 and -3. You are also welcome use the enumeration type `ECTSGrade` from an earlier exercise. The grade 2 is the lowest passing grade.

In both `BooleanCourse` and `GradedCourse` you should write a method called `Passed`. The method is supposed to return whether our sample student passes the course.

The class `Project` aggregates two boolean courses and two graded courses. You can assume that a project is passed if at least three out of the four courses are passed. Write a method `Passed` in class `Project` which implements this passing policy.

Make a project with four courses, and try out your solution.

In this exercise you are supposed to make a simple and rather primitive solution. We will come back to this exercise when we have learned about inheritance and collection classes.

11.10. Class Variables

Lecture 3 - slide 16

A class variable in a class C is shared between all instances (objects) of C. In addition, a class can be used even in the case where there does not exist any instance of C at all. Some classes are not intended to be instantiated. Such classes act as modules, cf. our discussion of modules in Section 2.3.

A *class variable* belongs to the class, and it is shared among all instances of the class.

- Class variables
  - are declared by use of the `static` modifier in C#
  - may be used as *global variables* - associated with a given class
  - do typically hold *meta information* about the class, such as the number of instances

In Program 11.10 we show a new version of the `BankAccount` class, in which there is a private, static variable `nextAccountNumber` of type `long`. When we make a `BankAccount` object, we give it a unique account number. The output, which is shown in Listing 11.12, is produced by a client similar to Program 11.6. The program output reveals the effect of the static variable `nextAccountNumber`.

```
using System;

public class BankAccount {

   private double interestRate;
   private string owner;
   private decimal balance;
   private long accountNumber;

   private static long nextAccountNumber = 0;

   public BankAccount(string owner) {
      nextAccountNumber++;
      this.accountNumber = nextAccountNumber;
      this.interestRate = 0.0;
      this.owner = owner;
      this.balance = 0.0M;
   }

   public BankAccount(string owner, double interestRate) {
      nextAccountNumber++;
      this.accountNumber = nextAccountNumber;
      this.interestRate = interestRate;
      this.owner = owner;
      this.balance = 0.0M;
   }

   // Some methods not shown in this version

   public override string ToString() {
      return owner + "'s account, no. " + accountNumber + " holds " +
            + balance + " kroner";
   }
}
```

Program 11.10 *The sketch of class BankAccount with a class variable.*

```
Kurt's account, no. 1 holds 100 kroner
Bent's account, no. 2 holds 1030 kroner
Thomas's account, no. 3 holds 3060 kroner
```

Listing 11.12 *Output of the BankAccount client program.*

Exercise 3.7. *Sharing the Random Generator*

In the `Die` class shown in the start of this lecture, each `Die` object creates its own `Random` object. (If you access this exercise from the web edition there are direct links to the relevant versions of class `Die` and class `Random`).

We observed that tosses of two or more instances of class `Die` will be identical. Explain the reason of this behavior.

Modify the `Die` class such that all of them share a single `Random` object. Consider different ways to implement this sharing. Rerun the `Die` program and find out if "the parallel tossing pattern" observed

above has been alleviated.

11.11. Class Methods

Lecture 3 - slide 17

Class methods are not connected to any instance of a class. Thus, class methods can be activated without providing any instance of the class. A class method `M` in a class `C` is activated by `C.M(...)`. The tree dots stand for possible actual parameters.

The static method `Main` plays a particular role in a C# program, because the program execution starts in `Main`. (Notice that `Main` starts with a capital M). It is crucial that `Main` is static, because there are objects around at the time `Main` is called. Thus, it is not possible to activate any instance method at that point in time! We have seen `Main` used many times already. There can be a `Main` method in more than one class. `Main` is either parameter less, or it may take an array of strings (of type `String[]`).

A *class method* is associated with the class itself, as opposed to an object of the class

- A class method **`M`** in a class **`C`**
  - is declared by use of the **`static`** modifier in C#
  - can only access static members of the class
  - must be activated on the class as such
  - is activated as **`C.M(...)`** from outside **`C`**
  - can also be activated as **`M(...)`** from inside C

In order to illustrate the use of static methods in C# we extend Program 11.10 with a couple of static methods, see line 32-41 of Program 11.13. The static method `GetAccount` is the most interesting one. It searches the static `accounts` variable (of type `ArrayList`) for an account with a given number. It returns the located bank account if it is found. If not, it returns `null`. Notice the way the `GetAccount` method is used in Program 11.14.

```
1  using System;
2  using System.Collections;
3
4  public class BankAccount {
5
6     private double interestRate;
7     private string owner;
8     private decimal balance;
9     private long accountNumber;
10
11    private static long nextAccountNumber = 0;
12    private static ArrayList accounts = new ArrayList();
13
14    public BankAccount(string owner) {
15       nextAccountNumber++;
16       accounts.Add(this);
17       this.accountNumber = nextAccountNumber;
18       this.interestRate = 0.0;
19       this.owner = owner;
20       this.balance = 0.0M;
21    }
22
```

```
23    public BankAccount(string owner, double interestRate) {
24       nextAccountNumber++;
25       accounts.Add(this);
26       this.accountNumber = nextAccountNumber;
27       this.interestRate = interestRate;
28       this.owner = owner;
29       this.balance = 0.0M;
30    }
31
32    public static long NumberOfAccounts (){
33      return nextAccountNumber;
34    }
35
36    public static BankAccount GetAccount (long accountNumber){
37       foreach(BankAccount ba in accounts)
38         if (ba.accountNumber == accountNumber)
39            return ba;
40       return null;
41    }
42
43    // Some BankAccount methods are not shown in this version
44
45 }
```

Program 11.13 *A sketch of a BankAccount class with static methods.*

```
1  using System;
2
3  public class BankAccountClient {
4
5    public static void Main(){
6      BankAccount a1 = new BankAccount("Kurt", 0.02),
7                  a2 = new BankAccount("Bent", 0.03),
8                  a3 = new BankAccount("Thomas", 0.02);
9
10     a1.Deposit(100.0M);
11     a2.Deposit(1000.0M); a2.AddInterests();
12     a3.Deposit(3000.0M); a3.AddInterests();
13
14     BankAccount a = BankAccount.GetAccount(2);
15     if (a != null)
16       Console.WriteLine(a);
17     else
18       Console.WriteLine("Cannot find account 2");
19   }
20
21 }
```

Program 11.14 *A client BankAccount.*

When we run Program 11.14 we get the output shown in Listing 11.15 (only on web).

In Program 11.16 we show an example of a typical error. I bet that you will experience this error many times yourself. Can you see the problem? If not, read the text below Program 11.16.

```
1  using System;
2
3  public class BankAccountClient {
4
5    BankAccount
6      a1 = new BankAccount("Kurt", 0.02),    // Error:
7      a2 = new BankAccount("Bent", 0.03),    // An object reference is
8      a3 = new BankAccount("Thomas", 0.02);  // required for the
```

```
9                                                  // nonstatic field
10   public static void Main(){
11
12     a1.deposit(100.0);
13     a2.deposit(1000.0); a2.addInterests();
14     a3.deposit(3000.0); a3.addInterests();
15
16     Console.WriteLine(a1);
17     Console.WriteLine(a2);
18     Console.WriteLine(a3);
19   }
20
21 }
```

Program 11.16 *A typical problem: A class method that accesses instance variables.*

The variables `a1`, `a2`, and `a3` in Program 11.16 are instance variable of class `BankAccountClient`. Thus, these variables are used to hold the state of objects of type `BankAccountClient`. The problem is that there does not exist any object of type `BankAccountClient`. We only have the class `BankAccountClient`. Therefore we need to declare `a1`, `a2`, and `a3` as static. Alternatively, we can rearrange the program such that `a1`, `a2`, and `a3` become local variables of the `Main` method. As yet another alternative, we can instantiate the class `BankAccountClient`, and move the body of `Main` to an instance method. The latter alternative is illustrated in Program 11.17.

11.12. Static Classes and Partial Classes in C#

Lecture 3 - slide 18

A *static* class C can only have static members

A *partial* class is defined in two or more source files

- **Static class**
  - Serves as a *module* rather than a *class*
  - Prevents instantiation, subclassing, instance members, and use as a type.
  - Examples: `System.Math`, `System.IO.File`, and `System.IO.Directory`
- **Partial class**
  - Usage: To combine manually authored and automatically generated class parts.

It is possible to use the modifier 'static' on a class. A class marked as `static` can only have static members, and it cannot be instantiated. A static class is similar to a sealed class (see Section 30.4) which we do not (or cannot) instantiate. However, a static class is more restrictive, because it also disallows instance members, and it cannot be used as a type in field declarations and in method parameter lists.

There are some pre-existing C# classes that exclusively contain static methods. The class `System.Math` is such a class. It contains mathematical constants such as *e* and *pi*. It also contains commonly used mathematical functions such as `Abs`, `Cos`, `Sin`, `Log`, and `Exp`. It would be strange (and therefore illegal) to attempt an instantiation of such a class.

The static classes `File` and `Directory` in the namespace `System.IO` are discussed in Chapter 38.

A partial class, marked with the `partial` modifier, can be used if it is practical to aggregate a class from more than one source file. This is, in particular, handy when a class is built from automatically generated parts and manually authored parts (such as a GUI class). Use of partial classes may also turn out to be handy when a group of programmers participate in the programming of a single, large class.

11.13. Constant and readonly variables

Lecture 3 - slide 19

The variables we have seen until now can be assigned to new values at any time during the program execution. In this section we will study variable with no or limited assignment possibilities. Of obvious reasons, it is confusing to call these "variables". Therefore we use the term "constant" instead.

C# supports two different kinds of constants. Some constants, denoted with the `const` modifier, are bound at compile time. Others, denoted with the `readonly` modifier, are bound at object creation time.

Constants and readonly variables cannot be changed during program execution

- *Constants* declared with use of the **`const`** keyword
  - *Computed at compile-time*
  - Must be initialized by an initializer
  - The initializer is evaluated at compile time
  - No memory is allocated to constants
  - Must be of a simple type, a string, or a reference type
- *Readonly variables* declared with use of the **`readonly`** modifier
  - *Computed at object-creation time*
  - Must either be initialized by an initializer or in a constructor
  - Cannot be modified in other parts of the program

It can be noticed that compile-time bound constants can only be of simple types, `string`, or a reference type. In addition, for non-string reference types, the only possible value is `null`.

Program 11.17 demonstrate some legal uses of constant and readonly variables. The elements emphasized with **green** are all legal and noteworthy. Notice first that we in `Main` instantiates the `ConstDemo` class, such that we can work on instance variables, as opposed to (static) class variables.

In line 4 we bind the constant `ca` to 5.0 and the constant `cb` to 6.0. This is done by the compiler, before the program starts executing. Notice that the compiler can carry out simple computations, as in line 5. In line 7 and 8 we bind the readonly variables `roa` and `rob` to 7.0 and to the value of the expression `Log(e)`. It is possible to assign to `roa` and `rob` in the constructor, but after the execution of the constructor `roa` and `rob` are non-assignable. In line 11 we assign `roba` to a new `BankAccount`. Notice that it - in addition - is legal to assign to read-only variables in constructors (line 14 and 15). This is - on the other hand - the last possible, legal assignments to `roa` and `roba`. In line 24 we see that we can mutate a bank account despite that the account is referred by a readonly variable. We modify the object, not the variable that references the object.

```
1  using System;
2
3  class ConstDemo {
4    const double    ca = 5.0,
```

```
5                   cb = ca + 1;
6
7   private readonly double roa = 7.0,
8                           rob = Math.Log(Math.E);
9
10  private readonly BankAccount
11                  roba = new BankAccount("Anders");
12
13  public ConstDemo(){   // CONSTRUCTOR
14    roa = 8.0;
15    roba = new BankAccount("Tim");
16  }
17
18  public static void Main(){
19    ConstDemo self = new ConstDemo();
20    self.Go();
21  }
22
23  public void Go(){
24    roba.Deposit(100.0M);
25  }
26}
```

Program 11.17 *Legal use of constants and readonly variables.*

Program 11.18 domonstrates a number of illegal uses of constants and readonly variables. The elements emphasized with **red** are all illegal. The compiler catches all of them. In line 12 and 21 we attempt an assignment to the (compile-time) constant `ca`. This is illegal - even in a constructor. In line 22 and 23 we see that it is illegal to assign to readonly variables, such as `roa` and `roba`, once they have been initialized.

```
1 using System;
2
3 class ConstDemo {
4   const double    ca = 5.0;
5
6   private readonly double roa = 7.0;
7
8   private readonly BankAccount
9                   roba = new BankAccount("Anders");
10
11  public ConstDemo(){   // CONSTRUCTOR
12    ca = 6.0;
13  }
14
15  public static void Main(){
16    ConstDemo self = new ConstDemo();
17    self.Go();
18  }
19
20  public void Go(){
21    ca = 6.0;
22    roa = 8.0;
23    roba = new BankAccount("Peter");
24  }
25}
```

Program 11.18 *Illegal use of constant and readonly variables.*

11.14. Objects and Classes

Lecture 3 - slide 20

At an overall level (as for instance in OOA and OOD) objects are often characterized in terms of *identity*, *state*, and *behavior*. Let us briefly address each of these, and relate them to programming concepts.

An object has an *identity* which makes it different and distinct from any other object. Two objects which are created by two activations of the `new` operator never share identity (they are not identical). In the practical world, the identity of an object is associated to its location in the memory: its address. Two objects are identical if their addresses are the same. But be careful here. The address of an object is not necessarily fixed and constant through the life time of the object. The object may be moved around in the memory of the computer, without losing its identify.

The *state* of the object corresponds to the data, as prescribed by the class to which the object belongs. As such, the state pertains to the instance variables of the class, see Section 11.8.

The *behavior* of the object is prescribed by the operations of the class, to which the object belongs. We have already discussed instance methods in Section 11.9. In Chapter 18 through Chapter 23 we will discuss operations, and hereby object behavior, in great details.

We practice *object-oriented programming*, but we write classes in our programs. This may be a little confusing. Shouldn't we rather talk about *class-oriented programming*?

When we write an object-oriented program, we are able to program all (forthcoming) objects of a given type/class together. This is done by writing the class. Thus, we write the classes in our source programs, but we often imagine a (forthcoming) situation where the class "is an object" which interacts with a number of other objects - of the same type or of different types.

At run time, the class that we wrote, prescribes the behavior of all the objects which are instances of the class.

In our source program we deal with classes. The classes exist for a long time - typically years. In the running program we have objects. The objects exist while the program is running. A typical program runs a few seconds, minutes, or perhaps hours. Often, we want to preserve our objects in between program executions. This turns out to be a challenge! We discuss how to preserve objects with use of serialization in Section 39.1.

All objects cease to exist when the program execution terminates.

This is in conflict with the behavior of corresponding real-life phenomena, and it causes a lot of problems and challenges in many programs

There are no objects in the source programs! Only classes. You may ask if there are classes in the running program. It makes sense to represent the classes in the running program, such that we can access the classes as data. Most object-oriented systems today represent the classes as particular objects called *metaobjects*. This is connected to an area in computer science called *reflection*.

Classes are written and described in source programs

Objects are created and exist while programs are running

11.15. The current object - this

Lecture 3 - slide 21

We have earlier discussed the role of the current object, see Section 10.3.

The current object in a C# program execution is denoted by the variable **`this`**

`this` is used for several different purposes in C#:

- Reference to shadowed instance variables
- Activation of another constructor
- Definition of indexers
- In definition of extension methods

This use of **`this`** for access of shadowed instance variables has been used in many of the classes we have seen until now. For an example see line 10 of Program 11.2.

Use of **`this`** for activation of another constructor is, for instance, illustrated in line 10 and 14 of Program 12.4.

Use of **`this`** in relation to definition of indexers is discussed in Section 19.1, illustrated for instance in line 10 of Program 19.1.

11.16. Visibility Issues

Lecture 3 - slide 22

In this section we will clarify some issues that are related to visibility. We will, in particular, study a type of error which is difficult to deal with.

Let us first summarize some facts about visibility of types and members:

- Types in namespaces
  - Either public or internal
  - Default visibility: internal
- Members in classes
  - Either private, public, internal, protected or internal protected
  - Default visibility: private
- Visibility inconsistencies
  - A type T can be less accessible than a method that returns a value of type T

Below we will rather carefully explain the mentioned inconsistency problem.

In Program 11.19 we have shown an internal class `C` in a namespace `N`. As given in Program 11.19 `C` is only supposed to be used inside the namespace `N`. In reality we have forgotten to state that `C` is public in `N`. I every now and then forget the modifier "`public`" in front of "`class C`" (line 3). I guess that you will run into this problem too - sooner og later.

Based on the internal class `C` in the namespace `N` we will now describe a scenario that leads to an error that can be difficult to understand. The class `D` is also located in `N`, and therefore `D` can use `C`. Class `D` is public in `N`. (If `D` had been located in another namespace, it would not have access to class `C`). A method `M` in class `D` makes and returns a `C`-object.

We cannot compile the program just described. We get an "*inconsistent accessibility error*". The compiler tells you that the return type of method `M` (which is `C`) is less accessible than the method `M` itself. In other words, `M` returns an object of a type, which cannot be accessed.

The cure is to make the class `C` public in its namespace. Thus, just add a `public` modifier in front of "class C" in line 3 of Program 11.19.

```
1  namespace N{
2
3    class C {
4
5    }
6
7    public class D{
8
9      public C M(){      // Compiler-time error message:
10       return new C();
11                        // Inconsistent accessibility:
12                        // return type 'N.C' is less
13                        // accessible than method 'N.D.M()'
14     }
15
16   }
17
18 }
```

Program 11.19 *An illustration of the 'Inconsistent Accessibility' problem.*

Please notice this kind of compiler error, and the way to proceed when you get it. I have witnessed a prospective student programmer who used several days to figure out what the compiler meant with the "*inconsistent accessibility error*". Now you are warned!

11.17. References

[Goldberg83] Adele Goldberg and David Robson, *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[Meyer88] Bertrand Meyer, *Object-oriented software construction*. Prentice Hall, 1988.

[Tennent81] Tennent, R.D., *Principles of Programming Languages*. Prentice Hall, 1981.

12. Creating and Deleting Objects

In this chapter we will explore the creation of object from classes, and how to get rid of the objects once they are not longer needed. Creation of objects - instantiation of classes - is tightly connected with initialization of new objects. Object initialization is therefore also an important theme in this chapter.

12.1. Creating and Deleting Objects

Lecture 3 - slide 24

Our goal in this section is to obtain an *overall* understanding of object creation and deletion, in particular in relation to the dimension of explicit/implicit creation and deletion. If you dislike such overall discussion, please proceed to Section 12.2. We identify the following approaches to creation and deletion of objects:

- Creating Objects
  - By instantiating classes
    - Implicitly: via variable declarations
    - Explicitly: on demand, by command
  - By copying existing object - *cloning*
- Deleting Objects
  - Explicitly: on demand, by command
  - Implicitly: deleted when not used any longer - via use of *garbage collection*

The most important way to create objects is to instantiate a class. Instantiation takes place when we use the class as a template for creating a new object. We may have an explicit way to express this (such as the operator `new`), or it may be implicitly done via declaration of a variable of the type of the class. Relative to this understanding, C# uses explicit creation of objects from classes, and implicit creation of objects (values) from structs.

Instantiation is the process of allocating memory to a new object of some class

Instantiation comes in two flavors:

- **Static instantiation:**
  - The object is automatically created (and destroyed) when the surrounding object or block is created.
- **Dynamic instantiation:**
  - The object is created on demand, by calling a particular operator (`new`).

Static instantiation is implicit. The object is automatically created (and destroyed) when the surrounding object or block is created. *Dynamic instantiation* is explicit. The object is created on demand, by executing a command. In C# and similar language we call the `new` operator for the purpose of dynamic class instantiation.

We should also be aware of the possibility of object copying. If we already have a nice object, say `obj`, we can create a new object (of the same type as `obj`) by copying `obj`. Some object-oriented programming

languages (most notably Self) use this as the only way of creating objects. The original objects in Self are called *prototypes*, and they are created directly by the programmer (instead of classes).

Older object-oriented programming languages, such as C++, use explicit deleting of objects. Most newer object-oriented programming languages use implicit object deleting, by means of garbage collection. The use of garbage collection turns out to be a major quality of an object-oriented programming language. C# relies on garbage collection.

> Modern object-oriented languages support explicit object creation and implicit object deletion (by means of garbage collection)

12.2. Instantiation of classes in C#

Lecture 3 - slide 26

We illustrate instantiation of classes in C# using a client of a `Point` class, such as Program 11.2, or even better a similar class with non-public instance variables. The accompanying slide shows such a class.

> Classes must be instantiated dynamically with use of the **new** operator
>
> The **new** operator returns a reference to the new object

The class `Application` in Program 12.1 uses class `Point`. Recall that class `Application` is said to be a client of class `Point`. We have three `Point` variables `p0`, `p1`, and `p2`. The two latter variables are local variables in `Main`. `p0` is static, because it is used from a static method.

We see a single instantiation of class `Point` at the **purple** place. `p0` is automatically initialized to `null` and `p1` is uninitialized before the assignments `p0 = p1 = p2`. After the assignments all three variables refer to the same `Point` object , and therefore you should be able to understand the program output shown in Listing 12.2. Notice the `Move` message in line 12 and the implementation of `Move` in line 13-15 of Program 11.2.

```
1  using System;
2
3  public class Application{
4
5    private static Point p0;   // Initialized to null
6
7    public static void Main(){
8      Point p1,                // NOT initialized
9            p2 = new Point(1.1, 2.2);
10
11     p0 = p1 = p2;
12     p2.Move(3.3, 0);
13     Console.WriteLine("{0} {1} {2}", p0, p1, p2);
14   }
15
16 }
```

Program 12.1 *Use of the class Point in a client class called Application.*

`Move` in line 12 moves the object referred by the three variables `p0`, `p1`, and `p2`. If you have problems with this, you are encouraged to review this example when you have read Section 13.2.

```
1 Point: (4,4, 2,2).  Point: (4,4, 2,2).  Point: (4,4, 2,2).
```

Listing 12.2 *Output from the Point client program.*

12.3. Initialization of objects

Lecture 3 - slide 27

Initialization should always follow class instantiation.

Initialization is the process of ascribing initial values to the instance variables of an object

There are several ways to do initialization. We recommend that you are *explicit about initialization* in your programs. With use of explicit initialization you signal that you have actually thought about the initialization. If you rely on default values, it may equally well be the case that you have not considered the initialization at all!

Initialization of an object of type T can be done

- Via use of *default values of T*
  - *zero* for numeric types, *false* for `bool`, `'\x0000'` for `char`, and `null` for reference types
- Via use of an *initializer*
- Via special methods called *constructors*

In C# you can denote the default value of a type `T` by use of the expression `default(T)`. For a reference type `RT`, `default(RT)` is is `null`. For a value type `VT`, `default(VT)` is the default value of `VT`. The *default value* of numeric types is *zero*, the default value of `bool` is *false*, the default `char` value is the null character, and the default value of reference types is null. The default value of a struct type is aggregated by the default values of the fields of the struct.

In Program 12.1 we have seen that local variables are not initialized to the default value of their types. Instance variables (fields) in classes are, however. This is confusing, and it may easily lead to errors if you forget the exact rules of the language.

An initializer is, for instance, the expression following '=' in a declaration such as `int i = 5 + j;`

It is **not** recommended to initialize instance variables via initializers. Initializers are static code, and from static code you cannot refer to the current object, and you cannot refer to other instance variables.

You should write one or more constructors of every class, and you should explicitly initialize all instance variables in your constructors. By following this rule you do not have to care about default values.

It is very important that a newly born object is initialized to a healthy citizen in the population of objects

Explicit initialization is always preferred over implicit initialization

Always initialize instance variables in constructors

12.4. Constructors in C#

Lecture 3 - slide 28

As recommended in Section 12.3, initialization of instance variables takes place in constructors.

A *constructor* is a special method which is called automatically in order to initialize a new instance of a class

- Constructors in C#
  - Have the same name as the surrounding class
  - Do not specify any return value type
  - Are often overloaded - several different constructors can appear in a class
  - May - in a special way - delegate the initialization job to another constructor
  - In case no constructors are defined, there is a parameterless *default constructor*
    - As its only action, it calls the parameterless constructor in the superclass
  - In case a constructor is defined there will be no parameterless default constructor

There is no '`constructor`' keyword in C#. By the way, there is no '`method`' keyword either. So how do we recognize constructors? The answer is given in first two bullet points above: A constructor has the same name as the surrounding class, and it specifies no return type.

Overloading takes place if we have two constructors (or methods) of the same name. Overloaded constructors are distinguished by different types of parameters. In Program 11.5 there are three overloaded constructors. Overload resolution takes place at compile time. It means that a constructor used in `new C(...)` is determined and bound at compile time - not at run time.

The special delegation mentioned in bullet point four is illustrated by the difference between Program 12.3 and Program 12.4. In the latter, the two first constructors activate the third constructor. The third constructor in Program 12.4 is the most general one, because it can handle the jobs of the two first mentioned constructors as special cases. Notice the **`this(...)`** syntax in between the constructor head and body.

As already stressed, I recommend that you always supply at least one constructor in the classes you program. In that case, there will be no parameterless default constructor available to you. You can always, however, program a parameterless constructor yourself. The philosophy is that if you have started to program constructors in your class, you should finish the job. It is not sound to mix your own, "custom" constructors (which are based on a deep knowledge about the class) with the system's default initialization (based on very little knowledge of the class).

In Program 11.5 we have seen a `BankAccount` class with three constructors. In Program 12.4 we show another version of the `BankAccount` class, also with three constructors. In both versions of the class, the three

constructors reflect different ways to initialize a new bank account. They provide convenience to clients of the `BankAccount` class. Program 12.4 is better than Program 11.5 because there is less overlap between the constructors. Thus, Program 12.4 is easier to maintain than Program 11.5. (Just count the lines and compare). Make sure to program your constructors like in Program 12.4.

```
using System;

public class BankAccount {

   private double interestRate;
   private string owner;
   private decimal balance;

   public BankAccount(string owner):
     this(owner, 0.0, 0.0M) {
   }

   public BankAccount(string owner, double interestRate):
     this(owner, interestRate, 0.0M) {
   }

   public BankAccount(string owner, double interestRate,
                      decimal balance) {
      this.interestRate = interestRate;
      this.owner = owner;
      this.balance = balance;
   }

   // BankAccount methods here
}
```

Program 12.4 *Improved constructors in class BankAccount.*

We also show and emphasize the constructors in the `Die` class, which we meet in Program 10.1 of Section 10.1. Below, in Program 12.5, the first `Die` constructor call the second one, hereby making a six eyed die. Notice that the second `Die` constructor creates a new `Random` object. It is typical that a constructor in a class instantiates a number of other classes, which again may instantiate other classes, etc.

```
using System;

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

  public Die (): this(6){}

  public Die (int maxNumberOfEyes){
    randomNumberSupplier =
      new Random(unchecked((int)DateTime.Now.Ticks));
    this.maxNumberOfEyes = maxNumberOfEyes;
    numberOfEyes = NewTossHowManyEyes();
  }

  // Die methods here

}
```

Program 12.5 *Constructors in the class Die.*

12.5. Copy constructors

Lecture 3 - slide 29

Copy constructors can be used for making copies of existing objects. A copy constructor can be recognized by the fact that it takes a parameter of the same type as the class to which it belongs. Object copying is an intricate matter, because we will have to decide if the referred object should be copied too (shallow copying, deep copying, or something in between, see more details in Section 13.4 and Section 32.6).

It is sometimes useful to have a constructor that creates an identical copy of an existing object

In Program 12.6 we show the `Die` class with an emphasized copy constructor. Notice that the `Random` object is shared between the original `Die` and the copy of the `Die`. This is shallow copying.

```
1  using System;
2
3  public class Die {
4    private int numberOfEyes;
5    private Random randomNumberSupplier;
6    private readonly int maxNumberOfEyes;
7
8    public Die (Die d){
9      numberOfEyes = d.numberOfEyes;
10     randomNumberSupplier = d.randomNumberSupplier;
11     maxNumberOfEyes = d.maxNumberOfEyes;
12   }
13
14   public Die (): this(6){}
15
16   public Die (int maxNumberOfEyes){
17     randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
18     this.maxNumberOfEyes = maxNumberOfEyes;
19     numberOfEyes = randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
20   }
21
22   // Die methods here
23
24 }
```

Program 12.6 *The class Die with a copy constructor.*

The use of copy constructors is particularly useful when we deal with mutable objects

Objects are mutable if their state can be changed after the constructor has been called. It is often necessary to copy a mutable object. Why? Because of aliasing, an object may be referred from several different places. If the object is mutable, all these places will observe a change, and this is not always what we want. Therefore, we can protect against this by copying certain objects.

The observation from above is illustrated by means of an example - privacy leak - in Section 16.5.

12.6. Initialization of class variables

Lecture 3 - slide 30

It is too late - and not natural - to initialize class variables in ordinary constructors

Constructors initialize new instances of classes. Class instances are objects. Class variables (static fields) do not belong to any object. They belong to the class as such, but they can be used from instances of the class as well. Class variables can be useful even in the case where no instances of the class will ever be made.

Therefore we will need other means than constructors to initialize class variables in C#. Initialization of a class variable of type T takes place at class load time

- Via the *default value of type T*
- Via the *static field initializers*
- Via a *static constructor*

Initialization of class variable (static fields) `v` of type `T` takes place implicitly. The variable `v` is, at load time, bound the distinguished default value of type `T`.

A static initializer is the expression at the right-hand side of "=" in a static field declaration. In Program 12.7 we have emphasized four examples of static initializers from line 13 to 16. The static initializers are executed in the order of appearance at class load time.

In Program 12.7 we show a simple playing card class called `Card` in which we organize all spade cards, all heart cards, all club cards, and all diamond cards in static arrays. The arrays are created in static initializers from line 13 to 16. It is convenient to initialize the elements of the arrays in a for loops. The right place of these for loops is in a static constructor. We show a static constructor in line 18-25 of Program 12.7.

Notice in line 19 of Program 12.7 how we get access to all enumeration values in a given enumeration type `ET` by the expression `Enum.GetValues(typeof(ET))`.

```
1  using System;
2
3  public class Card{
4    public enum CardSuite { Spade, Heart, Club, Diamond};
5    public enum CardValue { Ace = 1, Two = 2, Three = 3, Four = 4, Five = 5,
6                            Six = 6, Seven = 7, Eight = 8, Nine = 9,
7                            Ten = 10, Jack = 11, Queen = 12, King = 13,
8                          };
9
10   private CardSuite suite;
11   private CardValue value;
12
13   public static Card[] allSpades = new Card[14];
14   public static Card[] allHearts = new Card[14];
15   public static Card[] allClubs = new Card[14];
16   public static Card[] allDiamonds = new Card[14];
17
18   static Card(){
19     foreach(CardValue cv in Enum.GetValues(typeof(CardValue))){
20       allSpades[(int)cv] = new Card(CardSuite.Spade, cv);
21       allHearts[(int)cv] = new Card(CardSuite.Heart, cv);
22       allClubs[(int)cv] = new Card(CardSuite.Club, cv);
```

```
      allDiamonds[(int)cv] = new Card(CardSuite.Diamond, cv);
    }
  }

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

  public CardSuite Suite{
    get { return this.suite; }
  }

  public CardValue Value{
    get { return this.value; }
  }

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

Program 12.7 *The class PlayingCard with a static constructor.*

We also show how the static arrays can be used, see Program 12.8 and the output of the program, see Listing 12.9 (only on web).

```
using System;

class Client{

  public static void Main(){
    foreach (Card c in Card.allSpades)
      Console.WriteLine(c);
  }

}
```

Program 12.8 *A client of class PlayingCard.*

We recommend explicit initialization of all variables in a class, including static variables. It is recommended to initialize all instance variables in (instance) constructors. Most static variables can and should be initialized via use of initializers, directly associated with their declaration. In some special cases it is convenient to do a systematic initialization of class variables, for instance in a for loop. This can be done in a static initializer.