25. Specialization of Classes

In this section the topic is *inheritance*. Inheritance represents an organization of classes in which one class, say B, is defined on top of another class, say A. Class B inherits the members of class A, and in addition B can define its own members.

Use of inheritance makes it possible to *reuse* the data and operations of a class A in several so-called *subclasses*, such as B, C, and D, without coping these data and operations in the source code. Thus, if we modify class A we have also implicitly modified B, C and D.

There are several different views and understandings of inheritance, most dominantly specialization and extension. But also words such as subtyping and subclassing are used. We start our coverage by studying the idea of specialization.

25.1. Specialization of Classes

Lecture 7 - slide 2

The idea of specialization was introduced in Section 3.4 when we studied concepts and phenomena. In Section 3.4 we defined a specialization as a more narrow concept than its generalization. We will, in this chapter, use the inspiration from specialization of concepts to introduce specialization of classes.

> Classes are regarded as *types*, and specializations as *subtypes*
>
> Specialization facilitates definition of new classes from existing classes on a sound conceptual basis

With specialization we nominate a subset of the objects in a type as a subtype. The objects in the subset are chosen such that they have "something in common". Typically, the objects in the subset are constrained in a certain way that set them apart from the surrounding set of objects.

We often illustrate the generalization/specialization relationship between classes or types in a tree/graph structure. See Figure 25.1. The arrow from B to A means that *B is a specialization of A* . Later we will use the same notation for the extended understanding that *B inherits from A*.

Figure 25.1 *The class B is a specialization of class A*

Below - in the dark blue definition box - we give a slightly more realistic and concrete definition of specialization. The idea of subsetting is reflected in the first element of the definition. The second element is, in reality a consequence of the subsetting. The last element stresses that some operations in the specialization can be redefined to take advantage of the properties that unite the objects/values in the specialization.

If a class B is a *specialization* of a class A then

- The instances of B is a subset of the instances of A
- Operations and variables in A are also present in B
- Some operations from A may be redefined in B

25.2. The extension of class specialization

Lecture 7 - slide 3

In Section 3.1 we defined the extension of a concept as the collection of phenomena that is covered by the concept. In this section we will also define the *extension* of a class, namely as the set of objects which are instances of the class or type.

We will now take a look at the extension of a specialized class/type. The subsetting idea from Section 25.1 can now be formulated with reference to the extension of the class.

The *extension* of a specialized class B is a subset of the *extension* of the generalized class A

The relationships between the extension of A and B can be illustrated as follows, using the well-known notation of *wenn diagrams*.

Figure 25.2 *The extension of a class A is narrowed when the class is specialized to B*

Let us now introduce the **is-a** relation between the two classes A and B:

- A B-object **is an** A-object
- There is a **is-a** relation between class A and B

The **is-a** relation characterizes specialization. We may even formulate an "is-a test" that tests if B is a specialization of A. The **is-a** relation can be seen as contrast to the **has-a** relation, which is connected to *aggregation*, see Section 3.3.

The **is-a** relation forms a contrast to the **has-a** relation

The **is-a** relation characterizes *specialization*

The **has-a** relation characterizes *aggregation*

We will be more concrete with the **is-a** relation and the **is-a** test when we encounter examples in the forthcoming sections.

25.3. Example: Bank Accounts

Lecture 7 - slide 4

In Figure 25.3 we give three classes that specialize the class `BankAccount`.

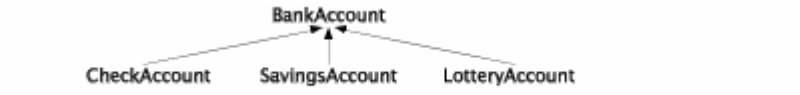

Figure 25.3 *A specialization hierarchy of bank accounts*

The **is-a** test confirms that there is a generalization-specialization relationship between `BankAccount` and `CheckAccount`: The statement "`CheckAccount` is a `BankAccount`" captures - very satisfactory - the relationships between the two classes. The statement "`BankAccount` is a `CheckAccount`" is not correct, because we can imagine bank accounts which are not related to checks at all.

As a contrast, the **has-a** test fails: It is against our intuition that "a `CheckAccount` has a `BankAccount`". Similarly, it is not the case that "`BankAccount` has a `CheckAccount`". Thus, the relationship between the classes `BankAccount` and `CheckAccount` is not connected to aggregation/decomposition.

In Figure 25.4 we show a possible constellation of extensions of the bank account classes. As hinted in the illustration, the specialized bank accounts overlap in such a way that there can exist a bank account which is both a `CheckAccount`, a `SavingsAccount`, and a `LotteryAccount`. An overlapping like in Figure 25.4 is the prerequisite for (conceptually sound) multiple specialization, see Section 27.5.

Figure 25.4 *Possible extensions of the bank account classes*

25.4. Example: Bank Accounts in C#

Lecture 7 - slide 5

In this section we show some concrete C# bank account classes, corresponding to the classes introduced in Figure 25.3.

The `BankAccount` class in Program 25.1 is similar to the class introduced earlier in Program 11.5. We need, however, to prepare for specialization/inheritance in a couple of ways. We briefly mention these preparations here. The detailed treatment will be done in the following sections.

First, we use protected instance variables instead of private instance variables. This allows the instance variables to be seen in the specialized bank account classes. See Section 27.3 for details.

Next, we use the virtual modifier for the methods that are introduced in class `BankAccount`. This allows these methods to be redefined in the specialized bank account classes. See Section 28.9.

```
1  using System;
2
3  public class BankAccount {
4
5     protected double interestRate;
6     protected string owner;
7     protected decimal balance;
8
9     public BankAccount(string o, decimal b, double ir) {
10       this.interestRate = ir;
11       this.owner = o;
12       this.balance = b;
13    }
14
15    public BankAccount(string o, double ir):
16      this(o, 0.0M, ir) {
17    }
18
19    public virtual decimal Balance {
20      get {return balance;}
21    }
22
23    public virtual void Withdraw (decimal amount) {
24       balance -= amount;
25    }
26
27    public virtual void Deposit (decimal amount) {
28       balance += amount;
29    }
30
31    public virtual void AddInterests() {
32       balance += balance * (Decimal)interestRate;
33    }
34
35    public override string ToString() {
36       return owner + "'s account holds " +
37              + balance + " kroner";
38    }
39 }
```

Program 25.1 *The base class BankAccount.*

The `CheckAccount` class shown in Program 25.2 redefines (overrides) the `Withdraw` method. This gives a special meaning to money withdrawal from a `CheckAccount` object. The method `ToString` is is also redefined (overridden) in class `CheckAccount`, in the same way as it was overridden in class `BankAccount` relative to its superclass (`Object`), see Program 25.1. Notice also the two constructors of class `CheckAccount`. They both delegate the construction work to `BankAccount` constructors via the **`base`** keyword. See Section 28.4 for details on constructors. This is similar to the delegation from one constructor to another in the same class, by use of **`this`**, as discussed in Section 12.4.

```
using System;

public class CheckAccount: BankAccount {

   public CheckAccount(string o, double ir):
     base(o, 0.0M, ir) {
   }

   public CheckAccount(string o, decimal b, double ir):
     base(o, b, ir) {
   }

   public override void Withdraw (decimal amount) {
      balance -= amount;
      if (amount < balance)
         interestRate = -0.10;
   }

   public override string ToString() {
      return owner + "'s check account holds " +
            + balance + " kroner";
   }
}
```

Program 25.2 *The class CheckAccount.*

The class `SavingsAccount` follow the same pattern as class `CheckAccount`. Notice that we also in class `SavingsAccount` redefine (override) the `AddInterests` method.

```
using System;

public class SavingsAccount: BankAccount {

   public SavingsAccount(string o, double ir):
     base(o, 0.0M, ir) {
   }

   public SavingsAccount(string o, decimal b, double ir):
     base(o, b, ir) {
   }

   public override void Withdraw (decimal amount) {
      if (amount < balance)
          balance -= amount;
      else
          throw new Exception("Cannot withdraw");
   }

   public override void AddInterests() {
      balance = balance + balance * (decimal)interestRate
                        - 100.0M;
   }

   public override string ToString() {
      return owner + "'s savings account holds " +
            + balance + " kroner";
   }
}
```

Program 25.3 *The class SavingsAccount.*

In the class `LotteryAccount` the method `AddInterests` is redefined (overridden). The idea behind a lottery account is that a few lucky accounts get a substantial amount of interests, whereas the majority of the accounts get no interests at all. This is provided for by the private instance variable `lottery`, which refers to a `Lottery` object. In the web-version of the material we show a definition of the `Lottery` class, which we program as a ***Singleton***.

```
1  using System;
2
3  public class LotteryAccount: BankAccount {
4
5     private static Lottery lottery  = Lottery.Instance(20);
6
7     public LotteryAccount(string o, decimal b):
8       base(o, b, 0.0) {
9     }
10
11    public override void AddInterests() {
12       int luckyNumber = lottery.DrawLotteryNumber;
13       balance = balance + lottery.AmountWon(luckyNumber);
14    }
15
16    public override string ToString() {
17       return owner + "'s lottery account holds " +
18              + balance + " kroner";
19    }
20 }
```

Program 25.4 *The class LotteryAccount.*

25.5. Example: Geometric Shapes

Lecture 7 - slide 6

In this section we show another example of specialization. The tree in Figure 25.5 illustrates a number of specializations of polygons. In the left branch of the tree we see the traditional and complete hierarchy of triangle types. In the right branch we show the most important specializations of quadrangles. Trapezoids are assumed to have exactly one pair of parallel sides, and as such trapezoids and parallelograms are disjoint.

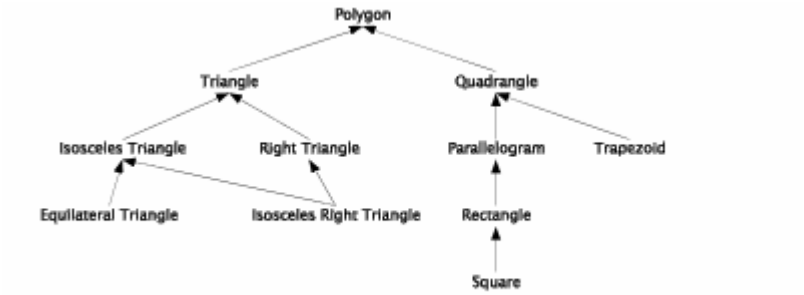

Figure 25.5 *A specialization hierarchy of polygons*

The polygon type hierarchy is a typical specialization hierarchy, because it fully complies with the definition of specialization from Section 25.1. The subset relationship is easy to verify. All operations defined at the polygon level are also available and meaningful on the specialized levels. In addition it makes sense to redefine many of the operations to obtain more accurate formula behind the calculations.

Overall, the deeper we come in the hierarchy, the more constraints apply. This is a typical characteristic of a real and pure generalization/specialization class hierarchy.

25.6. Specialization of classes

Lecture 7 - slide 7

We will now summarize the idea of class specialization. Objects of specialized classes

- fulfill stronger conditions (constraints) than objects of generalized classes
  - obey stronger *class invariants*
- have simpler and more accurate operations than objects of generalized classes

Specialization of classes in pure form do not occur very often.

Specialization in combination with extension is much more typical.

As noticed in Section 25.4 the hierarchy of polygons is real and pure example of specialization hierarchy.

The bank account hierarchy in Figure 25.3 is not as pure as the polygon hierarchy. The bank account hierarchy is - in the starting point - a specialization hierarchy, but the specialized classes are likely to be extended with operations, which do not make sense in the `BankAccount` class. Class extension is the topic in Chapter 26.

25.7. The Principle of Substitution

Lecture 7 - slide 8

The principle of substitution is described by Timothy Budd in section 8.3 of in his book *An Introduction to Object-oriented Programming* [Budd02]. The principle of substitution describes an ideal, which not always is in harmony with our practical and everyday programming experience. This corresponds to our observation that pure specialization only rarely is found in real-life programs.

If B is a subclass of A, it is possible to substitute an given instance of B in place of an instance of A without observable effect

As an example, consider the class hierarchy of polygons in Figure 25.5. Imagine that we have the following scene:

```
Polygon p = new Polygon(...);
RightTriangle tr = new RightTriangle(...);
/* Rest of program */
```

It is now possible to substitute the polygon object with the triangle object in the "rest of the program". This is possible because the triangle possesses all the general properties (area, circumference, etc) of the polygon. At least, the compiler will not complain, and the executing program will not halt. Notice, however, that the

substitution is only neutral to the actual meaning of the execution program if the replaced polygon actually happens to be the appropriate right triangle!

Notice that the opposite substitution does not always work. Thus, we cannot substitute a triangle with a general polygon (for instance a square). Most programs would break immediately if that was attempted. The reason is that a square does not, in general, possess the same properties as a triangle.

The ideas behind the principle of substitution are related to virtual methods (Section 28.14) and to dynamic binding (Section 28.11).

25.8. References

[Budd02] Timothy Budd, *An Introduction to Object-Oriented Programming*, third edition. Pearson. Addison Wesley, 2002.

26. Extension of Classes

Extension of classes is a more pragmatic concept than specialization of classes. Specialization of classes is directly based on - and inspired from - specialization of concepts, as discussed in Section 3.4. Extension of classes is a much more practical idea.

In the previous chapter (Chapter 25) we discussed specialization of classes. In this section we discuss class extension. In C# both class specialization and class extension will be dealt with by class inheritance, see Chapter 27.

26.1. Extension of Classes

Lecture 7 - slide 10

Classes can both be regarded as types and modules.

Class extension is a *program transport* and *program reusability* mechanism.

As the name suggests, class extension is concerned with adding something to a class. We can add both variables and operations.

We are not constrained in any way (by ideals of specialization or substitution) so we can in principle add whatever we want. However, we still want to have coherent and cohesive classes. We want classes focused on a single idea, where all data and operations are related to this idea. Our classes should be used as types for declaration of variables, and it should make sense to make instances of the classes. Thus, we do not want to treat classes are general purposes modules (in the sense of boxing modularity, see Section 2.3).

These considerations lead us to the following definition of *class extension*.

If class B is an *extension* of class A then

- B may add new variables and operations to A
- Operations and variables in A are also present in B
- B-objects are not necessarily conceptually related to A-objects

26.2. An example of simple extension

Lecture 7 - slide 11

In this section we will look at a typical example of class extension, which distinguishes itself from specialization as seen in Chapter 25.

Below, in Program 26.1 we show the class `Point2D`. It is a variant of one the `Point` types we have studied in Section 11.6, Section 14.3, and Section 18.2. The variant programmed below implements mutable points. This is seen in line 19, which assigns to the state of a `Point` object.

```
using System;

public class Point2D {
  private double x, y;

  public Point2D(double x, double y){
   this.x = x; this.y = y;
  }

  public double X{
    get {return x;}
  }

  public double Y{
    get {return y;}
  }

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

  public override string ToString(){
    return "Point2D: " + "(" + x + ", " + y + ")" + ".";
  }
}
```

Program 26.1 *The class Point2D.*

In Program 26.2 we extend the class `Point2D` with an extra coordinate, `z`, and hereby we get the class `Point3D`.

```
using System;

public class Point3D: Point2D {

  private double z;

  public Point3D(double x, double y, double z):
         base(x,y){
   this.z = z;
  }

  public double Z{
    get {return z;}
  }

  public void Move(double dx, double dy, double dz){
    base.Move(dx, dy);
    z += dz;
  }

  public override string ToString(){
    return "Point3D: " + "(" + X + ", " + Y + ", " + Z + ")" + ".";
  }
}
```

Program 26.2 *The class Point3D which extends class Point3d.*

Notice that `Move` in `Point3D` does not conflict with `Move` in `Point2D`. The reason is that the two methods are separated by the types of their formal parameters. The two `Move` operations in `Point3D` and `Point2D` are

(statically) overloaded. Thus relative to the discussion in Section 28.9 it is not necessary to supply a `new` modifier of `Move` in `Point3D`.

We also show how to use `Point2D` and `Point3D` in a client class, see Program 26.3. The output of the client program is shown in Listing 26.4.

```
1  using System;
2
3  public class Application{
4
5    public static void Main(){
6      Point2D p1 = new Point2D(1.1, 2.2),
7              p2 = new Point2D(3.3, 4.4);
8
9      Point3D q1 = new Point3D(1.1, 2.2, 3.3),
10             q2 = new Point3D(4.4, 5.5, 6.6);
11
12     p2.Move(1.0, 2.0);
13     q2.Move(1.0, 2.0, 3.0);
14     Console.WriteLine("{0} {1}", p1, p2);
15     Console.WriteLine("{0} {1}", q1, q2);
16   }
17
18 }
```

Program 26.3 *A client of the classes Point2D and Point3d.*

```
1 Point2D: (1,1, 2,2). Point2D: (4,3, 6,4).
2 Point3D: (1,1, 2,2, 3,3). Point3D: (5,4, 7,5, 9,6).
```

Listing 26.4 *The output from the Client program.*

The important observations about the extension `Point3D` of `Point2D` can be stated as follows:

- A 3D point is ***not*** a 2D point
- Thus, `Point3D` is not a specialization of `Point2D`
- The principle of substitution does not apply
- The set of 2D point objects is disjoint from the set of 3D points

The is-a test (see Section 25.2) fails on the class `Point3D` in relation to class `Point2D`. The "has-a test" also fails. It is not true that a 3 dimensional point has a 2 dimensional point as one its parts. Just look at the class `Point3D!` But - in reality - the "has-a test" is closer to success than the "is-a test". Exercise 7.1 researches an implementation of class `Point3D` in terms of a `Point2dD` part.

It is interesting to wonder if the principle of substitution applies, see Section 25.7. Can we substitute instances of `Point3D` in place of instances of `Point2D` without observable effects? Due to the independence and orthogonality of the three dimensions the principle of substitution is almost applicable. But the `Move` operation, as redefined in class `Point3D`, causes problems. The `Move` operation in class `Point2D` does an incomplete move when applied on a 3D point. And as noticed, `Move` in class `Point3D` is not a redefinition of `Move` from class `Point2D`. There are two different `Move` operations available on an instance of class `Point3D`. This is a mess!

In the last item it is stated that extensions (see Section 3.1) of class `Point2D` and class `Point3D` are disjoint (non-overlapping). Conceptually, there is no overlap between the set of two-dimensional points and the set of

three-dimensional points! This is probably - in a nutshell - the best in indication of the difference between the Point2D/Point3D example and - say - the `BankAccount` examples from Section 25.3 .

> The class Point2D was a convenient starting point of the class `Point3D`
>
> We have *reused* some data and operations from class `Point2D` in class `Point3D`

Exercise 7.1. *Point3D: A client or a subclass of Point2D?*

The purpose of this exercise is to sharpen your understanding of the difference between "*being a client of class C*" and "*being af subclass of class C*".

The class `Point3D` extends `Point2D` by means of inheritance.

As an alternative, the class `Point3D` may be implemented as a client of `Point2D`. In more practical terms this means that the class `Point3D` **has a**n instance variable of type `Point2D`. Now implement `Point3D` as a client of `Point2D` - such that a 3D point has a 2D point as a part.

Be sure that the class `Point3D` has the same interface as the version of class `Point3D` from the course material.

Evaluate the difference between "being a client of" an "extending" class `Point2D`. Which of the solutions do you prefer?

26.3. The intension of class extensions

Lecture 7 - slide 12

In Section 25.2 we realized that the essential characteristics of specialization is the narrowing of the class extension, see Figure 25.2. Above, in Section 26.2, we realized the the class extension of an extended class (such as `Point3D`) typically is disjoint from the class extension of the parent class (such as `Point2D`).

In this section we emphasize the similar, clear-cut characteristics of class extension, namely the enlargement of the class intension. This is illustrated in Figure 26.1.

> The *intension* of a class extension B is a superset of the *intension* of the original class A

Please be aware of possible confusion related to our terminology. We discuss class "extension" in this section, and we refer to the "intension" and "extension" (related to concepts, as discussed in Section 3.1). The two meanings of "extension" should be kept apart. They are used with entirely different meanings.

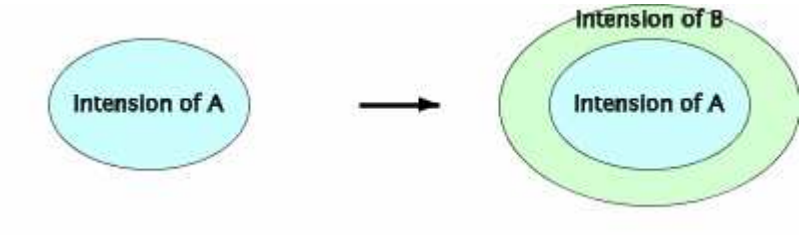

Figure 26.1 *The intension of a class A is blown up when the class is extended to B*

It is, in general, not possible to characterize the *extension* of B in relation to the *extension* of A

Often, the *extension* of A does not overlap with the *extension* of B

27. Inheritance in General

After we have discussed class specialization in Chapter 25 and class extension in Chapter 26 we will now turn our interest towards inheritance. Inheritance is a mechanism in an object-oriented programming language mechanism that supports both class specialization and class extension.

This section is about inheritance in general. Inheritance in C# is the topic of Chapter 28.

27.1. Inheritance

Lecture 7 - slide 14

When a number of classes inherit from each other a *class graph* is formed. If, for instance, both class B and C inherit from class A we get the graph structure in Figure 27.1. Later in this section, in Section 27.4, we will discuss which class graphs that make sense.

If a class B inherits the variables and operations from another class, A, we say that B is a *subclass* of A. Thus, in Figure 27.1 both B and C are subclasses of A. A is said to be a *superclass* of B and C.

Figure 27.1 *Two classes B and C that inherit from class A*

In the class graph shown in Figure 27.1 the edges are oriented from subclasses to superclasses. In other words, the arrows in the figure point at the common superclass.

In Figure 27.1 the members (variables and operations) of class A are also variables in class B and C, *just as though the variables and operations were defined explicitly in both class B and C*. In addition, class B and C can define variables and operations of their own. The inherited members from class A are not necessarily visible in class B and C, see Section 27.3. In essence, inheritance is a mechanisms that brings a number of variables and operations from the superclass to the subclasses.

Alternatively, we could copy the variables and operations from class A and paste them into class B and class C. This would, roughly, give the same result, but this approach is not attractive, and it should always be avoided. If we duplicate parts of our program it is difficult to maintain the program, because future program modifications must be carried out two or more places (both in class A, and in the duplications in class B and C). We always go for solutions that avoid such duplication of source program fragments.

When we run a program we make instances of our classes A, B and C. B and C have some data and operations that come from A (via inheritance). In addition, B and C have variables and operations of their own. Despite of this, an instance of class B is one single object, without any A part and B part. Thus, in an instance of class B the variables and operations of class A have been merged with the variables and operations from class B. In an instance of B there are very few traces left of the fact that class B actually inherits from class A.

The observations from above are summarized below. The situation described above, and illustrated in Figure 27.1

- Organizes the classes in a hierarchy
- Provides for some degree of specialization and/or extension of A
- At program development time, data and operations of A can be *reused* in B and C *without copying* and without any duplication in the source program
- At runtime, instances of class B and C are *whole objects*, without A parts

27.2. Interfaces between clients and subclasses

Lecture 7 - slide 15

The *client interface* of a class (say class A in Figure 27.2) is defined by the public members. This has been discussed in Section 11.1. In Figure 27.2 the client interface of class A is shown as number **1**.

The client interface of a class B (which is a subclass of class A) is extended in comparison with the client interface of class A itself. The client interface of class B basically includes the client interface of class A, and some extra definitions given directly in class B. The client interface of class B is shown as number **3** in Figure 27.2.

When inheritance is introduced, there is an additional kind of interface to take care of, namely the interfaces between a class and its subclasses. We call it the *subclass interface*. Interface number **2** in Figure 27.2 consists of all variables and operations in class A which are visible and hereby applicable in class B. Similarly, the interface numbered **4** is the interface between class B and its subclasses.

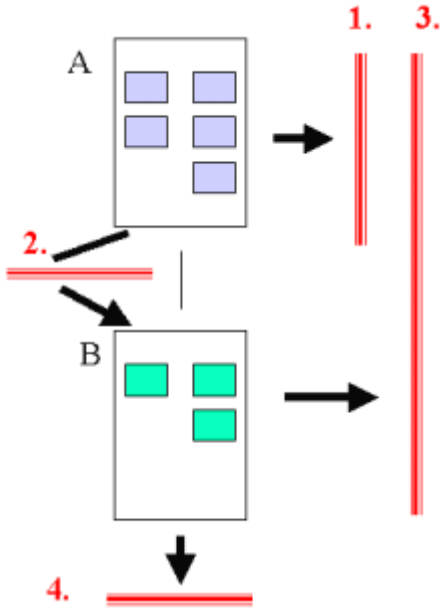

Figure 27.2 *Interfaces between A, B, their client classes, and their subclasses*

1. *The client interface of A*
2. *The subclass interface between A and its subclass B*
3. *The client interface of B*
4. *The subclass interface between B and potential subclasses of B*

27.3. Visibility and Inheritance

Lecture 7 - slide 16

Most object-oriented programming languages distinguish between private, protected and public variables and operations. Below we provide a general overview of these kinds of visibility.

- **Private**
  - Visibility limited to the class itself.
  - Instances of a given class can see each others private data and operations
- **Protected**
  - Visibility is limited to the class itself and to its subclasses
- **Public**
  - No visibility limitations

In Section 28.6 we refine the description of the visibility modifiers relative to C#.

27.4. Class hierarchies and Inheritance

Lecture 7 - slide 17

When a number of classes inherit from each other a class graph is defined. Class graphs were introduced in Section 27.1. Below we show different shapes of class graphs, and we indicate (by means of color and text) which of them that make sense.

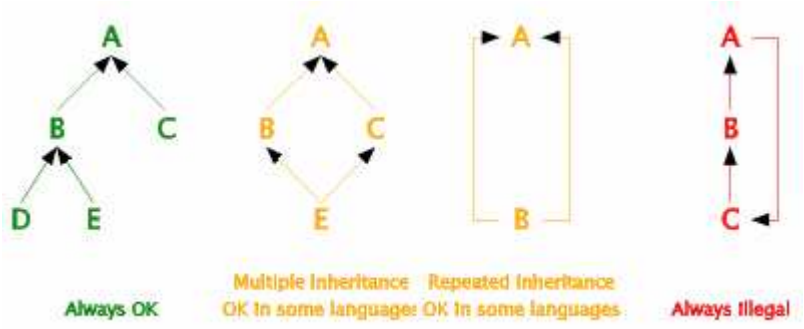

Figure 27.3 *Different graph structures among classes*

A tree-structured graph, as shown to the left in Figure 27.3 makes sense in all object-oriented programming languages. In Java and C# we can only construct tree structured class graphs. This is called *single-inheritance* because a class can at most have a single superclass.

Multiple inheritance is known from several object-oriented programming language, such as C++, Eiffel, and CLOS. Compared with single inheritance, multiple inheritance complicates the meaning of an object-oriented program. The nature of these complications will be discussed in Section 27.5.

Repeated inheritance is allowed more rarely. Eiffel allows it, however. It can be used to facilitate replication of superclass variables in subclasses.

Cyclic class graphs, as shown to the right in Figure 27.3 are never allowed.

27.5. Multiple inheritance

Lecture 7 - slide 18

In this section we dwell a little on multiple inheritance. Both relative to class specialization (see Chapter 25) and class extension (see Chapter 26) it can be argued that multiple inheritance is useful:

- Specialization of two or more classes
  - *Example:* An isosceles right triangle ***is a*** isosceles triangle and it ***is a*** right triangle
  - *Example:* There may exists a bank account which ***is a*** checking account and it ***is a*** savings account
- Extensions of two or more classes
  - "Program transport" from multiple superclasses

In Figure 25.4 the overlapping extensions of the classes `CheckAccount`, `SavingsAccount` and `LotteryAccount` indicate that there may exist a single object, which ***is a*** `CheckAccount`, a `SavingsAccount`, and a `LotteryAccount`.

When we in Section 26.2 discussed the extension of class `Point2D` to class `Point3D` it could have been the case that it was useful to extend class `Point3D` from an additional superclass as well.

Let us now briefly argue why multiple inheritance is difficult to deal with. In Figure 27.4 we have sketched a situation where class C inherits from both class A and class B. Both A and B have a variable or an operation named x. The question is now which x we get when we refer to x in C (for instance via C.x if x is static).

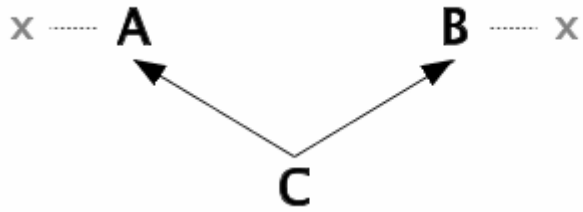

Figure 27.4 *Class B is a subclass of class A*

In general, the following problems and challenges can be identified:

- ***The name clash problem:*** Does x in C refer to the x in A or the x in B?
- ***The combination problem:*** Can x in A and x in B combined to a single x in C?
- ***The selection problem:*** Do we have means in C to select either x in A or x in B?
- ***The replication problem:*** Is there one or two x pieces in C?

Notice that some of these problems and challenges are slightly overlapping.

This ends the general discussion of inheritance. The next chapter is also about inheritance, as it relates to C#. The discussions of multiple inheritance is brought up again, in Chapter 31, in the context of interfaces.

28. Inheritance in C#

In Chapter 27 we discussed inheritance in general. In this section we will be more specific about class inheritance in C#. The current section is long, not least because it covers important details about virtual methods and polymorphism.

28.1. Class Inheritance in C#

Lecture 7 - slide 21

When we define a class, say `class-name`, we can give the name of the superclass, `super-class-name`, of the class. The syntax of this is shown in Chapter 27. In some contexts, a superclass is also called a base class.

```
class-modifier class class-name: super-class-name{
  declarations
}
```

Syntax 28.1 *A C# class defined as a subclass of given superclass*

We see that the superclass name is given after the colon. There is no keyword involved (like `extends` in Java). If a class implements interfaces, see Chapter 31, the names of these interfaces are also listed after the colon. The superclass name must be given before the names of interfaces. If we do not give a superclass name after the colon, it is equivalent to writing `: Object`. In other words, a class, which does not specify an explicit superclass, inherits from class `Object`. We discuss class `Object` in Section 28.2 and Section 28.3.

In Program 28.1 below we show a class `B` which inherits from class `A`. Notice that Program 28.1 uses C# syntax, and that the figure shows full class definitions. Notice also that the set of member is empty in both class `A` and `B`. As before, we use the graphical notation in Figure 28.1 for this situation.

```
1 class A {}
2
3 class B: A {}
```

Program 28.1 *A class A and its subclass B.*

Figure 28.1 *The class B inherits from class A*

B is said to be a *subclass* of A, and A a *superclass* of B. A is also called the *base class* of B.

28.2. The top of the class hierarchy

Lecture 7 - slide 22

As discussed in Section 27.4 a set of classes define a class hierarchy. The top/root of the class hierarchy is the class called `Object`. More precisely, the only class which does not have an edge to a superclass in the

class graph is called `Object`. In C# the class `Object` resides in the `System` namespace. The type `object` is an alias for `System.Object`. Due to inheritance the methods in class `Objects` are available in all types in C#, including value types. We enumerate these methods in Section 28.3.

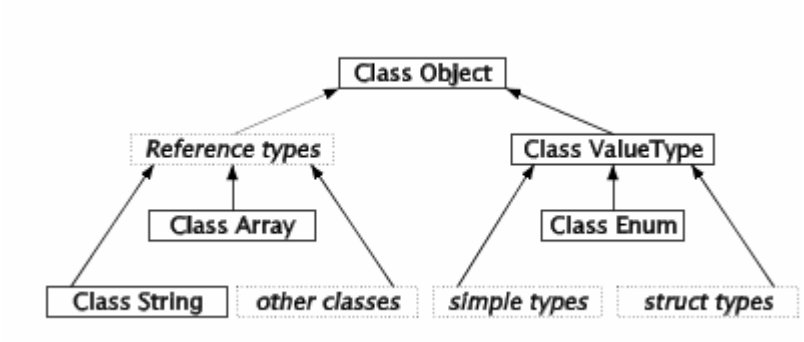

Figure 28.2 *The overall type hierarchy in C#*

The left branch of Figure 28.2 corresponds to the reference types of C#. Reference types were discussed in Chapter 13. The right branch of Figure 28.2 corresponds to the value types, which we have discussed in Chapter 14.

All pre-existing library classes, and the classes we define in our own programs, are reference types. We have also emphasized that strings (as represented by class `String`) and arrays (as represented by class `Array`) are reference types. Notice that the dotted box "Reference types" is imaginary and non-existing. (We have added it for matters of symmetry, and for improved conceptual overview). The role of class `Array` is clarified in Section 47.1.

The class `ValueType` is the base type of all value types. Its subclass `Enum` is a base type of all enumeration types. It is a little confusing that these two classes are used as superclasses of structs, in particular because structs cannot inherit from other structs or classes. This can be seen as a special-purpose organization, made by the C# language designers. We cannot, as programmers, replicate such organizations in our own programs. The classes `Object`, `ValueType` and `Enum` contain methods, which are available in the more specialized value types (defined by structs) of C#.

28.3. Methods in the class `Object` in C#

Lecture 7 - slide 23

We will now review the methods in class `Object`. Due to the type organization discussed in Section 28.2 these methods can be used uniformly in all classes and in all structs.

- Public methods in class **`Object`**
  - Equals:
    - **`obj1.Equals(obj2)`** - Instance method
    - **`Object.Equals(obj1, obj2)`** - Static method
    - **`Object.ReferenceEquals(obj1,obj2)`** - Static method
  - **`obj.GetHashCode()`**
  - **`obj.GetType()`**
  - **`obj.ToString()`**
- Protected methods in class **`Object`**
  - **`obj.Finalize()`**
  - **`obj.MemberwiseClone()`**

There are three equality methods in class `Object`. All three of them have been discussed in Section 13.5. The instance methods `Equals` is the one we often redefine in case we need a shallow equality operation in one of our classes. See Section 28.16 for details. The static method, also named `Equals`, is slightly more applicable because it can also compare objects/values and `null` values. The static method `ReferenceEquals` is - at least in the starting point - equivalent to the == operator.

The instance method `GetHashCode` produces an integer value which can be used for indexing purposes in hashtables. In order to obtain efficient implementations, `GetHashCode` often use some of the bit-wise operators, such as shifting and bit-wise exclusive or. (See Program 28.29 for an example). It must be ensured that if `o1.Equals(o2)` then `o1.GetHashCode()` has the same value as `o2.GetHashCode()`.

The instance method `ToString` is well-known. We have seen it in numerous types, for instance in the very first `Die` class we wrote in Program 10.1. We implement and override this method in most of our classes. `ToString` is implicitly called whenever we need some text string representation of an object `obj`, typically in the context of an output statement such `Console.WriteLine("{0}", obj)`. If the parameterless `ToString` method of class `Object` is not sufficient for our formatting purposes, we can implement the `ToString` method of the interface `IFormattable`, see Section 31.7.

The method `Finalize` is not used in C#. Instead, destructors are used. Destructors help release resources just before garbage collection is carried out. We do not discuss destructors in this material.

`MemberwiseClone` is a protected method which does bit per bit copying of an object (shallow copying, see Section 13.4). `MemberwiseClone` can be used in subclasses of `Object` (in all classes and structs), but `MemberwiseClone` cannot be used from clients because it is not public. In Section 32.7 we will see how to make cloning available in the client interface; This involves implementation of the interface `ICloneable` (see Section 31.4) and delegation to `MemberwiseClone` from the `Clone` method prescribed by `ICloneable`.

28.4. Inheritance and Constructors

Lecture 7 - slide 24

Constructors in C# were introduced in Section 12.4 as a means for initializing objects, cf. Section 12.3. It is recommended to review the basic rules for definition of constructors in Section 12.4.

As the only kind of members, constructors are not inherited. This is because a constructor is only useful in the class to which it belongs. In terms of the `BankAccount` class hierarchy shown in Figure 25.3, the `BankAccount` constructor is not directly useful as an inherited member of the class `CheckAccount`: It would not be natural to apply a `BankAccount` constructor on a `CheckAccount` object.

On the other hand, the `BankAccount` constructor typically does part of the work of a `CheckAccount` constructor. Therefore it is useful for the `CheckAccount` constructor to call the `BankAccount` constructor. This is indeed possible in C#. So the statement that "*constructors are not inherited*" should be taken with a grain of salt. A superclass constructor can be seen and activated in a subclass constructor.

Here follows the overall guidelines for constructors in class hierarchy:

- Each class in a class hierarchy should have its own constructor(s)
- The constructor of class C cooperates with constructors in superclasses of C to initialize a new instance of C
- A constructor in a subclass will always, implicitly or explicitly, call a constructor in its superclass

As recommended in Section 12.4 you should always program the necessary constructors in each of your classes. As explained and motivated in Section 12.4 it is not possible in C# to mix a parameterless *default constructor* and the constructors with parameters that you program yourself. You can, however, program your own parameterless constructor and a number of constructors with parameters.

In the same way as two or more constructors in a given class typically cooperate (delegate work to each other using the special `this(...)` syntax) the constructors of a class C and the constructors of the base class of C cooperate. If a constructor in class C does not explicitly call `base(...)` in its superclass, it implicitly calls the parameterless constructor in the superclass. In that case, such a parameterless constructor must exist, and it must be non-private.

We will return to the `BankAccount` class hierarchy from Section 25.4 and emphasize the constructors in the classes that are involved.

In Program 28.2 we see the root bank account class, `BankAccount`. It has two constructors, where the second is defined by means of the first. Notice the use of the `this(...)` notation outside the body of the constructor in line 16.

```
1  using System;
2
3  public class BankAccount {
4
5     protected double interestRate;
6     protected string owner;
7     protected decimal balance;
8
9     public BankAccount(string o, decimal b, double ir) {
10       this.interestRate = ir;
11       this.owner = o;
12       this.balance = b;
13    }
14
15    public BankAccount(string o, double ir):
16      this(o, 0.0M, ir) {
17    }
18
```

```
19    public virtual decimal Balance {
20      get {return balance;}
21    }
22
23    public virtual void Withdraw (decimal amount) {
24       balance -= amount;
25    }
26
27    public virtual void Deposit (decimal amount) {
28       balance += amount;
29    }
30
31    public virtual void AddInterests() {
32       balance += balance * (Decimal)interestRate;
33    }
34
35    public override string ToString() {
36       return owner + "'s account holds " +
37              + balance + " kroner";
38    }
39 }
```

Program 28.2 *Constructors in class BankAccount.*

The two constructors of the class `CheckAccount`, shown in Program 28.3, both delegate part of the initialization work to the first constructor in class `BankAccount`. Again, this is done via the special notation `base(...)` outside the body of the constructor. Notice that bodies of both constructors in `CheckAccount` are empty.

It is interesting to ask why the designers of C# have decided on the special way of delegating work between constructors in C#. Alternatively, one constructor could chose to delegate work to another constructor inside the bodies. The rationale behind the C# design is most probably, that the designers insist on a particular initialization order. This will be discussed in Section 28.5.

```
1  using System;
2
3  public class CheckAccount: BankAccount {
4
5     public CheckAccount(string o, double ir):
6       base(o, 0.0M, ir) {
7     }
8
9     public CheckAccount(string o, decimal b, double ir):
10      base(o, b, ir) {
11    }
12
13    public override void Withdraw (decimal amount) {
14       balance -= amount;
15       if (amount < balance)
16          interestRate = -0.10;
17    }
18
19    public override string ToString() {
20       return owner + "'s check account holds " +
21              + balance + " kroner";
22    }
23 }
```

Program 28.3 *Constructors in class CheckAccount.*

In the web-version of the material we also show the classes `SavingsAccount` and `LotteryAccount`, see Program 28.4 and Program 28.5 respectively.

28.5. Constructors and initialization order

Lecture 7 - slide 25

We speculated about the motives behind the special syntax of constructor delegation in the previous section. A constructor in a subclass must - either implicitly or explicitly - activate a constructor in a superclass. In that way a chain of constructors are executed when an object is initialized. The chain of constructors will be called from the most general to the least general. The following initializations take place when a new `C` object is made with `new C(...)`:

- Instance variables in `C` are initialized (field initializers)
- Instance variables in superclasses are initialized - most specialized first
- Constructors of the superclasses are executed - most general first
- The constructor body of **`C`** is executed

Notice that initializers are executed first, from most specific to most general. Next the constructors are called in the opposite direction.

Let us illustrate this by means of concrete example in Program 28.6, Program 28.7 and Program 28.8 where class `C` inherits from class `B`, which in turn inherit from class `A`.

The slightly artificial class `Init`, shown in Program 28.9 contains a static "tracing method" which returns a given init value, `val`. More importantly, for our interests, it tells us about the initialization. In that way we can see the initialization order on the standard output stream. The tiny application class, containing the static `Main` method, is shown in Program 28.10.

The output in Listing 28.11 reveals - as expected - that all initializers are executed before the constructors. First in class `C`, next in `B`, and finally in `A`. After execution of the initializers the constructors are executed. First the `A` constructors, then the `B` constructor, and finally the `C` constructor.

```
1  using System;
2
3  public class C: B {
4    private int varC1 = Init.InitMe(1, "varC1, initializer in class C"),
5                varC2;
6
7    public C (){
8      varC2 = Init.InitMe(4, "VarC2, constructor body C");
9    }
10 }
```

Program 28.6 *Initializers and constructors of class C.*

```
1  using System;
2
3  public class B: A {
4    private int varB1 = Init.InitMe(1, "varB1, initializer in class B"),
5                varB2;
6
7    public B (){
```

```
8       varB2 = Init.InitMe(4, "VarB2, constructor body B");
9     }
10  }
```

Program 28.7 *Initializers and constructors of class B.*

```
1   using System;
2
3   public class A {
4     private int varA1 = Init.InitMe(1, "varA1, initializer in class A"),
5                 varA2;
6
7     public A (){
8       varA2 = Init.InitMe(4, "VarA2, constructor body A");
9     }
10  }
```

Program 28.8 *Initializers and constructors of class A.*

```
1   using System;
2
3   public class Init{
4
5     public static int InitMe(int val, string who){
6       Console.WriteLine(who);
7       return val;
8     }
9
10  }
```

Program 28.9 *The class Init and the method InitMe.*

```
1   using System;
2
3   class App{
4
5     public static void Main(){
6       C c = new C();
7     }
8   }
```

Program 28.10 *A program that instantiates and initializes class C.*

```
1   varC1, initializer in class C
2   varB1, initializer in class B
3   varA1, initializer in class A
4   VarA2, constructor body A
5   VarB2, constructor body B
6   VarC2, constructor body C
```

Listing 28.11 *The output that reveals the initialization order.*

28.6. Visibility modifiers in C#

Lecture 7 - slide 27

Visibility control is a key issue in object-oriented programming. The general discussion about visibility appears in Section 11.3, Section 11.4 and Section 11.5. The C# specific discussion is briefly touched on in Section 11.7. We gave overview of visibility in namespaces and types in Section 11.16. In this lecture we have briefly described the issue in general in Section 27.3.

Basically, we must distinguish between visibility of types in assemblies and visibility of members in types:

- Visibility of a type (e.g. a class) in an assembly
  - **`internal`**: The type is not visible from outside the assembly
  - **`public`**: The type is visible outside the assembly
- Visibility of members in type (e.g., methods in classes)
  - **`private`**: Accessible only in the containing type
  - **`protected`**: Accessible in the containing type and in subtypes
  - **`internal`**: Accessible in the assembly
  - **`protected internal`:** Accessible in the assembly and in the containing type and its subtypes
  - **`public`**: Accessible whenever the enclosing type is accessible

The issue of inheritance and visibility of private members is addressed in Exercise 7.2.

Internal visibility is related to assemblies, not namespaces. Assemblies are produced by the compiler, and represented as either `-.dll` or `-.exe` files. It is possible to have a type which is invisible outside the assembly, into which it is compiled. It is, of course, also possible to have types which are visible outside the assembly. This is the mere purpose of having libraries. Per default - if you do not write any modifier - top-level types are internal in their assembly. The ultimate visibility of members of a class, quite naturally, depends on the visibility of the surrounding type in the assembly.

Members of classes (variables, methods, properties, etc) can also have internal visibility. Protected members are visible in direct and indirect subclasses. You can think of protected members as members visible from classes in the inheritance family. We could call it *family visibility*. It is - as noticed above - possible to combine internal and protected visibility. The default visibility of members in types is private.

It was a major point in Chapter 11 that data should be private within its class. With the introduction of inheritance we may chose to define data as protected members. Protected data is convenient, at least from a short-term consideration, because superclass data then can be seen from subclasses. But having protected data in class C implies that knowledge of the data representation is spread from class C to all direct and indirect subclasses of C. Thus, a larger part of the program is vulnerable if/when the data representation is changed. (Recall the discussion about *representation independence* from Section 11.6). Therefore we may decide to keep data private, and to access superclass data via public or protected operations. It is worth a serious consideration is you should allow protected data in the classes of your next programming project.

Related to inheritance we should also notice that a redefined member in a subclass should be at least as visible as the member in the superclass, which it replaces. It is possible to introduce *visibility inconsistencies*. This has been discussed in great details in Section 11.16.

Exercise 7.2. *Private Visibility and inheritance*

Take a look at the classes shown below:

```
using System;

public class A{
  private int i = 7;
```

```
  protected int F(int j){
   return i + j;
  }
}

public class B : A{
  public void G(){
    Console.WriteLine("i: {0}", i);
    Console.WriteLine("F(5): {0}", F(5));
  }
}

public class Client {
  public static void Main(){
    B b = new B();
    b.G();
  }
}
```

Answer the following questions before you run the program:

1. Does the instance of `B`, created in `Main` in `Client`, have an instance variable `i`?
2. Is the first call to `Console.WriteLine` in `G` legal?
3. Is the second call to `Console.WriteLine` in `G` legal?

Run the program and confirm your answers.

Exercise 7.3. *Internal Visibility*

The purpose of this exercise is to get some experience with the visibility modifier called **internal**. Take a look at the slide to which this exercise belongs.

In this exercise, it is recommended to activate the compiler from a command prompt.

Make a namespace `N` with two classes `P` and `I`:

- `P` should be public. `P` should have a static public member `p` and a static internal member `i`.
- `I` should be internal. `I` should also have a static public member `p` and a static internal member `i`.

Compile the classes in the namespace `N` to a single assembly, for instance located in the file x.dll.

Demonstrate that the class `I` can be used in class `P`. Also demonstrate that `P.i` can be seen and used in class `P`.

After this, program a class `A`, which attempts to use the classes `P` and `I` from x.dll. Arrange that class `A` is compiled separately, to a file y.dll. Answer the following questions about class `A`:

1. Can you declare variables of type `P` in class `A`?
2. Can you declare variables of type `I` in class `A`?
3. Can you access `P.i` and and `P.p` in `A`?
4. Can you access `I.i` and and `I.p` in `A`?

Finally, arrange that class `A` is compiled together with `N.P` and `N.I` to a single assembly, say y.dll. Does this alternative organization affect the answers to the questions asked above?

28.7. Inheritance of methods, properties, and indexers

Lecture 7 - slide 28

All members apart from constructors are inherited. In particular we notice that operations (methods, properties, and indexers) are inherited.

Methods, properties, and indexers are inherited

Here follows some basic observations about inheritance of operations:

- Methods, properties, and indexers can be redefined in two different senses:
  - Same names and signatures in super- and subclass, closely related meanings (`virtual`, `override`)
  - Same names and signatures in super- and subclass, two entirely different meanings (`new`)
- A method M in a subclass B can refer to a method M in a superclass A
  - `base.M(...)`
  - Cooperation, also known as method combination

The distinctions between `virtual`/`override` and `new` is detailed in Section 28.9.

The subject of the second item is method combination, which we will discuss in more details in Chapter 29.

Operators are inherited. A redefined operator in a subclass will be an entirely new operator.

Operators (see Chapter 21) are static. The choice of operator is fully determined at compile time. Operators can be overloaded. There are rules, which constrain the types of formal parameters of operators, see Section 21.4. All this implies that two identically named operators in two classes, one of which inherits from the other, can be distinguished from each other already at compile-time.

28.8. Inheritance of methods: Example.

Lecture 7 - slide 29

We will now carefully explore a concrete example that involves class inheritance. We stick to the bank account classes, as introduced in Section 25.4 where we discussed class specialization. In Program 28.12, Program 28.13, and Program 28.14 we emphasize the relevant aspects of inheritance with colors.

```
1  using System;
2
3  public class BankAccount {
4
5     protected double interestRate;
```

```
6     protected string owner;
7     protected decimal balance;
8
9     public BankAccount(string o, decimal b, double ir) {
10       this.interestRate = ir;
11       this.owner = o;
12       this.balance = b;
13    }
14
15    public BankAccount(string o, double ir):
16      this(o, 0.0M, ir) {
17    }
18
19    public virtual decimal Balance {
20      get {return balance;}
21    }
22
23    public virtual void Withdraw (decimal amount) {
24       balance -= amount;
25    }
26
27    public virtual void Deposit (decimal amount) {
28       balance += amount;
29    }
30
31    public virtual void AddInterests() {
32       balance += balance * (Decimal)interestRate;
33    }
34
35    public override string ToString() {
36       return owner + "'s account holds " +
37              + balance + " kroner";
38    }
39 }
```

Program 28.12 *The base class BankAccount.*

In Program 28.12 the data a protected, not private. This is an easy solution, but not necessarily the best solution, because the program area that uses the three instance variables of class `BankAccount` now becomes much larger. This has already been discussed in Section 28.6. In addition the properties and methods are declared as virtual. As we will see in Section 28.14 this implies that we can redefine the operations in subclasses of `BankAccount`, such that the run-time types of bank accounts (the dynamic types) determine the actual operations carried out.

```
1  using System;
2
3  public class CheckAccount: BankAccount {
4
5     // Instance variables of BankAccount are inherited
6
7     public CheckAccount(string o, double ir):
8       base(o, 0.0M, ir) {
9     }
10
11    public CheckAccount(string o, decimal b, double ir):
12      base(o, b, ir) {
13    }
14
15    // Method Balance is inherited
16    // Method Deposit is inherited
17    // Method AddInterests is inherited
18
```

```
19    public override void Withdraw (decimal amount) {
20       base.Withdraw(amount);
21       if (amount < balance)
22          interestRate = -0.10;
23    }
24
25    public override string ToString() {
26       return owner + "'s check account holds " +
27             + balance + " kroner";
28    }
29 }
```

Program 28.13 *The class CheckAccount.*

In class `CheckAccount` in Program 28.13 the instance variables of class `BankAccount` and the operations `Balance`, `Deposit`, and `AddInterests` are inherited. Thus, these operations from `BankAccount` can simply be (re)used on `CheckAccount` objects. The method `Withdraw` is redefined. Notice that `Withdraw` calls `base.Withdraw`, the `Withdraw` method in class `BankAccount`. This is (imperative) method combination, see Section 29.1. As we will see in Section 28.9 the modifier **`override`** is crucial. The method `ToString` overrides the similar method in `BankAccount`, which in turn override the similar method from class `Object`.

In the web-version of the material we also show subclasses `SavingsAccount` and `LotteryAccount`.

Exercise 7.4. *A subclass of LotteryAccount*

On the slide, to which this exercise belongs, we have emphasized inheritance of methods and properties in the bank account class hierarchy. From the web-version of the material there is direct access to the necessary pieces of program.

The `LotteryAccount` uses an instance of a `Lottery` object for adding interests. Under some lucky circumstances, the owner of a `LotteryAccount` will get a substantial amount of interests. In most cases, however, no interests will be added.

There exists a single file which contains the classes `BankAccount`, `CheckAccount`, `SavingsAccount`, `Lottery`, together with a sample client class.

Program a specialization of the `LotteryAccount`, called `LotteyPlusAccount`, with the following redefinitions of `Deposit` and `Withdraw`.

- The `Deposit` method doubles the deposited amount in case you draw a winning lottery number upon deposit. If you are not lucky, `Deposit` works as in `LottoryAccount`, but an administrative fee of 15 kroner will be withdrawn from your `LotteryPlusAccount`.
- The `Withdraw` method returns the withdrawn amount without actually deducting it from the `LotteryPlusAccount` if you draw a winning lottery number upon withdrawal. If you are not lucky, `Withdraw` works as in `LottoryAccount`, and an additional administrative fee of 50 kroner will be withdrawn from the account as well.

Notice that the `Deposit` and `Withdraw` methods in `LotteryPlusAccount` should combine with the method in `LotteryAccount` (method combination). Thus, use the `Deposit` and `Withdraw` methods from `LotteryAccount` as much as possible when you program the `LotteryPlusAccount`.

Test-drive the class `LotteryPlusAccount` from a sample client class.

28.9. Overriding and Hiding in C#

Lecture 7 - slide 30

Let us now carefully explore the situation where a method `M` appears in both class `A` and its subclass `B`. Thus, the situation is as outlined in Program 28.16.

```
1 class A {
2   public void M(){}
3 }
4
5 class B: A{
6   public void M(){}
7 }
```

Program 28.16 *Two methods M in classes A and B, where B inherits from A.*

Let us already now reveal that Program 28.16 is illegal in C#. The compiler will complain (with a warning). We will need to add some modifiers in front of the method definitions.

There are basically two different situations that make sense:

- **Intended redefinition:**
  B.M is intended to redefine A.M - such that B.M is used on B instances
  - A.M must be declared as **`virtual`**
  - B.M must be declared to **`override`** A.M
- **Accidental redefinition:**
  The programmer of class B is not aware of A.M
  - B.M must declare that it is not related to A.M - using the **`new`** modifier

Intended redefinition is - by far - the most typical situation. We prepare for intended redefinition by declaring the method as **`virtual`** in the most general superclass. This causes the method to be virtual in all subclasses. Each subclass that redefines the method must **`override`** it. This pattern paves the road for dynamic binding, see Section 28.10. Intended redefinition appears frequently in almost all object-oriented programs. We have already seen it several times in the bank account classes in Program 28.12 - Program 28.15.

Accidental redefinition is much more rare. Instead of declaring `M.B` as **`new`** it is better to give `M` in `B` another name. The **`new`** modifier should only be used in situations where renaming is not possible nor desirable.

28.10. Polymorphism. Static and dynamic types

Lecture 7 - slide 31

In this section we define the concepts of polymorphism and dynamic binding. In order to be precise about dynamic binding we also define the meaning of static and dynamic types of variables and parameters.

Polymorphism stands for the idea that a variable can refer to objects of several different types

The *static type* of a variable is the type of variable, as declared

The *dynamic type* of a variable is type of object to which the variable refers

Dynamic binding is in effect if the dynamic type of a variable `v` determines the operation activated by `v.op(...)`

'*Poly*' means 'many' and '*morph*' means 'form'. Thus, polymorphism is related to the idea of 'having many forms' or 'having many types'. In the literature, polymorphism is often associated with procedures or functions that can accept parameters of several types. This is called *parametric polymorphism*. More basically (and as advocated by, for instance, Bertrand Meyer [Meyer88]), polymorphism can be related to variables. A polymorphic variable or parameter can (at run-time) take values of more than one type. This is called *data polymorphism*.

A concrete and detailed discussion of dynamic and static types, based on an example, is found in Section 28.11, which is the next section of this material.

Use of the modifiers `virtual` and `override`, as discussed in Section 28.9 is synonymous with dynamic binding. We have much more to say about dynamic binding later in this material, more specifically in Section 28.14 and Section 28.15. Polymorphism and good use of dynamic binding is one of the *"OOP crown jewels"* in relation to inheritance. It means that you should attempt to design your programs such that they take advantage of polymorphism and dynamic binding. For a practical illustration, please compare Program 28.26 and Program 28.27 in Section 28.15.

28.11. Static and dynamic types in C#

Lecture 7 - slide 32

Before we can continue our discussion of virtual methods (dynamic binding) we will give examples of static and dynamic types of variables.

We now apply the definitions from Section 28.10 to the scene in Program 28.17 shown below. As it appears, the class `B` inherits from class `A`. In the client of `A` and `B` the variable `x` is declared of type `A`, and the variable `y` is declared of type `B`. In other words, the static type of `x` is `A` and the static type of `y` is `B`.

Next, in line 10 and 11, we instantiate class `A` and `B`. Thus, at the position of line 12, the variable `x` refers to an object of type `A`, and the variable `y` refers to an object of type `B`. Therefore, at the position of line 12, the dynamic type of `x` is `A` and the dynamic type of `y` is `B`.

The assignment `x = y` in line 13 implies that `x` (as well as `y`) now refer to a `B` object. This is possible due polymorphism. Recall that a `B` object **is an** `A` object. You can read about the **is-a** relation in Section 25.2.

Line 15 causes a compile-time error. The variable `y`, of static type `B`, cannot refer an object of type `A`. An instance of class `A` **is not a** `B` object.

Finally, in line 17, we assign `x` to `y`. Recall, that just before line 17 `x` and `y` refer to the same `B` object. Thus, the assignment `y = x` is harmless in the given situation. Nevertheless, it is illegal! From a general and

conservative point of view, the danger is that the variable y of static type `B` can be assigned to refer to an object of type `A`. This would be illegal, because an `A` object is (still) not a `B` object.

```
1  class A {}
2  class B: A{}
3
4  class Client{
5    public static void Main (){
6      //           // Static type    Dynamic type
7      A x;         //    A             -
8      B y;         //    B             -
9
10     x = new A(); //    A             A   TRIVIAL
11     y = new B(); //    B             B   TRIVIAL
12
13     x = y;       //    A             B   OK - TYPICAL
14
15     y = new A(); //    B             A   Compile time ERROR
16                  //                      Cannot implicitly convert type 'A' to 'B'.
17     y = x;       //    B             B   Compile time ERROR !
18                  //                      Cannot implicitly convert type 'A' to 'B'.
19   }
20 }
```

Program 28.17 *Illustration of static and dynamic types.*

We will now, in Program 28.18 remedy one of the problems that we encountered above in Program 28.17. In line 16 the assignment `y = x` succeed if we cast the object, referred to by `x`, to a `B`-object. You should think of the cast as a way to assure the compiler that `x`, at the given point in time, actually refers to a `B`-object.

In line 15 we attempt a similar cast of the object returned by the expression `new A()`. (This is an attempted downcast, see Section 28.17). As indicated, this causes a run-time error. It is not possible to convert an A object to a B object.

```
1  class A {}
2  class B: A{}
3
4  class Client{
5    public static void Main (){
6      //             // Static type    Dynamic type
7      A x;           //    A             -
8      B y;           //    B             -
9
10     x = new A();   //    A             A   TRIVIAL
11     y = new B();   //    B             B   TRIVIAL
12
13     x = y;         //    A             B   OK - TYPICAL
14
15     y = (B)new A(); //    B            A   RUNTIME ERROR
16     y = (B)x;      //    B             B   NOW OK
17   }
18 }
```

Program 28.18 *Corrections of the errors in the illustration of static and dynamic types.*

With a good understanding of static and dynamic types of variables you can jump directly to Section 28.14. If you read linearly you will in Section 28.12 and in Section 28.13 encounter the means of expressions in C# for doing type testing and type conversion.

28.12. Type test and type conversion in C#

Lecture 7 - slide 33

It is possible to test if the dynamic type of a variable `v` is of type `C`, and there are two ways to convert (cast) one class type to another

The following gives an overview of the possibilities.

- **`v is C`**
  - True if the variable **`v`** is of *dynamic type* **`C`**
  - Also true if the variable **`v`** is of dynamic type **`D`**, where **`D`** is a subtype of **`C`**

As it appears from level 9 of Table 6.1 **`is`** an operator in C#. - The explanation of the **`is`** operator above is not fully accurate. The expression in the item above is true if `v` successfully can be converted to the type `C` by a reference conversion, a boxing conversion, or an unboxing conversion.

It is - every now and then - useful to test the dynamic type of a variable (or expression) by use of the **`is`** operator. Notice however, that in many contexts it is unnecessary to do so explicitly. Use of a virtual method (dynamic binding) encompasses an implicit test of the dynamic type of an expression. Such a test is therefore an implicit branching point in a program. In other words, passing a message to an object selects an appropriate method on basis of the type of the receiver object. You should always consider twice if it is really necessary to discriminate with use of the **`is`** operator. If your program contains a lot of instance tests (using the **`is`** operator) you may not have understood the idea of virtual methods!

The following to forms of type conversion (casting) is supported in C#:

- **`(C)v`**
  - Convert the *static type* of **`v`** to **`C`** in the given expression
  - Only possible if the dynamic type of **`v`** is **`C`** , or a subtype of **`C`**
  - If not, an `InvalidCastException` is thrown
- **`v as C`**
  - Non-fatal variant of **`(C)v`**
  - Thus, convert the static type of **`v`** to **`C`** in the given expression
  - Returns `null` if the dynamic type of **`v`** is not **`C`**, or a subtype of **`C`**

The first, `(C)v`, is know as *casting*. If **`C`** is a class, casting is a way to adjust the static type of a variable or expression. The latter alternative, `v as C`, is equivalent to `(C)v` provided that no exceptions are thrown. If `(C)v` throws an exception, the expression `v as C` returns `null`.

Above we have assumed that `C` is a reference type (a class for instance). It also makes sense to use `(T)v` where `T` is value type (such as a struct). In this case a value of the type is converted to another type. We have touched on explicitly programmed type conversions in Section 21.2. See an example in Program 21.3. Casting of a value of value type may change the actual bits behind the value. The casting of a reference, as discussed above, does not change the "bits behind the reference".

as is an operator in the same way as **is**, see level 9 of Table 6.1. Notice also, at level 13 of the table, that casting is an operator in C#.

> The **typeof** operator can be applied on a typename to obtain the corresponding object of class `Type`
>
> The **`Object.GetType`** instance method returns an object of class `Type` that represents the run-time type of the receiver.

Examples of casting, and examples of the **as** and **is** operators, are given next in Section 28.13.

28.13. Examples of type test and type conversion

Lecture 7 - slide 34

In the web-version of the material, this section contains concrete examples that show how to use the **is**, **as**, and typecasting operators. The examples are relatively large, and the explanations quite detailed; Therefore they have been left out of the paper edition.

Exercise 7.5. *Static and dynamic types*

Type conversion with **`v as T`** was illustrated with a program on the accompanying slide. The output of the program was confusing and misleading. We want to report the static types of the expressions `ba1 as BankAccount`, `ba1 as CheckAccount`, etc. If you access this exercise from the web-version there will be direct links to the appropriate pieces of program.

Explain the output of the program. You can examine the classes `BankAccount`, `CheckAccount`, `SavingsAccount` and `LotteryAccount`, if you need it.

Modify the program such that the static type of the expressions **`bai as Banktype`**`Account` is reported. Instead of

```
baRes1 = ba1 as BankAccount;
Report(baRes1);
```

you should activate some method on the expression `ba1 as BankAccount` which reveals its static type. In order to do so, it is allowed to add extra methods to the bank account classes.

28.14. Virtual methods in C#

Lecture 7 - slide 35

This section continues our discussion of dynamic binding and virtual methods from Section 28.10. We will make good use of the notion of static type and dynamic type, as introduced in Section 28.11.

First of all notice that virtual methods that are overridden in subclasses rely on dynamic binding, as defined in Section 28.10. Also notice that everything we tell about virtual methods also holds for virtual properties and virtual indexers.

The ABC example in Program 28.24 shows two classes, `A` and `B`, together with a `Client` class. `B` is a subclass of `A`. The class `A` holds the methods `M`, `N`, `O`, and `P` which are redefined somehow in the subclass `B`.

The compiler issues a warning in line 11 because we have a method `M` in both class `A` and class `B`. Similarly, a warning is issued in line 13 because we have a method `O` in class `B` as well as a virtual method `O` in class `A`. The warnings tells you that you should either use the modifier `override` or `new` when you redefine methods in class `B`.

`M` in class `B` is said to *hide* `M` in class `A`. Similarly, `O` in class `B` *hides* `O` in class `A`.

The overriding of `N` in line 12 (in class `B`) of the virtual method `N` in line 5 (from class `A`) is very typical. Below, in the client program, we explain the consequences of this setup. Please notice this pattern. Object-oriented programmers use it again and again. It is so common that it is the default setup in Java!

The method `P` in line 14 of class `B` is declared as `new`. `P` in class `B` hides `P` in class `A`. The use of `new` suppresses the warnings we get for method `M` and for method `O`. The use of `new` has nothing to do with class instantiation. Declaring `P` as `new` in `B` states an accidental name clash between methods in the class hierarchy. `P` in `A` and `P` in `B` can co-exist, but they are not intended to be related in the same way as `N` in `A` and `N` in `B`.

```
1  using System;
2
3  class A {
4    public void          M( ){Console.WriteLine("M in A");}
5    public virtual void  N( ){Console.WriteLine("N in A");}
6    public virtual void  O( ){Console.WriteLine("O in A");}
7    public void          P( ){Console.WriteLine("P in A");}
8  }
9
10 class B: A{
11   public void          M( ){Console.WriteLine("M in B");} // warning
12   public override void N( ){Console.WriteLine("N in B");}
13   public void          O( ){Console.WriteLine("O in B");} // warning
14   public new void      P( ){Console.WriteLine("P in B");}
15 }
16
17 class Client {
18   public static void Main(){
19     A aa = new A( ),     // aa has static type A, and dynamic type A
20       ab = new B( );     // ab has static type A, and dynamic type B
21     B b = new B( );      // b  has static type B, and dynamic type B
22
23     aa.N( );   ab.N( );   b.N( );    // The dynamic type controls
24     Console.WriteLine( );
25     aa.P( );   ab.P( );   b.P( );    // The static type controls
26   }
27 }
```

Program 28.24 *An illustration of virtual and new methods in class A and B.*

The `Client` class in Program 28.24 brings objects of class `A` and `B` in play. The variable `aa` refers an `A` object. The variable `ab` refers a `B` object. And finally, the variable `b` refers a `B` object as well.

The most noteworthy cases are emphasized in **blue**. When we call a virtual method `N`, the dynamic type of the receiving object controls which method to call. Thus in line 23, `aa.N()` calls the `N` method in class `A`, and `ab.N()` calls the `N` method in class `B`. In both cases we *dispatch* on an object referred from variables of static type `A`. The dynamic type of the variable controls the dispatching.

In line 25, the expression `aa.P()` calls the `P` method in class `A`, and (most important in this example) `ab.P()` also class the `P` method in class `A`. In both cases the static type of the variables `aa` and `ab` control the dispatching. Please consult the program output in Listing 28.25 to confirm these results.

```
1 N in A
2 N in B
3 N in B
4
5 P in A
6 P in A
7 P in B
```

Listing 28.25 *Output from the program that illustrates virtual and new methods.*

Virtual methods use dynamic binding

Properties and indexers can be virtual in the same way as methods

Let us finally draw the attention to the case where a virtual method `M` is overridden along a long chain of classes, say `A`, `B`, `C`, `D`, `E`, `F`, `G`, and `H` that inherit from each other (`B` inherits from `A`, `C` from `B`, etc). In the middle of this chain, let us say in class `E`, the method `M` is defined as **`new virtual`** instead of being overridden. This changes almost everything! It is easy to miss the **`new virtual`** method among all the overridden methods. If a variable `v` of static type `A`, `B`, `C`, or `D` refers to an object of type `H`, then `v.M()` refers to `M` in D (the level just below the **`new virtual`** method). If `v` is of static type `E`, `F`, or `G` then `v.M()` refers to `M` in class `H`.

28.15. Practical use of virtual methods in C#

Lecture 7 - slide 36

Having survived the ABC example from the previous section, we will now look at a real-life example of virtual methods. We will program a client class of different types of bank account classes, and we will see how the `AddInterests` method benefits from being virtual.

The bank account classes, used below, were introduced in Section 25.4 in the context of our discussion of specialization. Please take a look at the way the `AddInterests` methods are defined in Program 25.1, Program 25.3, and Program 25.4. The class `CheckAccount` inherits the `AddInterests` method of class `BankAccount`. `SavingsAccount` and `LotteryAccount` override `AddInterests`.

Notice that the definition of the `AddInterests` methods follow the pattern of the methods named `N` in Program 28.24.

```
1  using System;
2
3  public class AccountClient{
4
5    public static void Main(){
6       BankAccount[] accounts =
7        new BankAccount[5]{
8          new CheckAccount("Per",1000.0M, 0.03),
9          new SavingsAccount("Poul",1000.0M, 0.03),
10         new CheckAccount("Kurt",1000.0M, 0.03),
11         new LotteryAccount("Bent",1000.0M),
12         new LotteryAccount("Lone",1000.0M)
13       };
14
15     foreach(BankAccount ba in accounts){
16       ba.AddInterests();
17     }
18
19     foreach(BankAccount ba in accounts){
20       Console.WriteLine("{0}", ba);
21     }
22   }
23
24 }
```

Program 28.26 *Use of virtual bank account methods.*

The `Main` method of the `AccountClient` class in Program 28.27 declares an array of type `BankAccount`, see line 6. Due to polymorphism (see Section 28.10) it is possible to initialize the array with different types of `BankAccount` objects, see line 7-13.

We add interests to all accounts in the array in line 15-17. This is done in a **foreach** loop. The expression `ba.AddInterests()` calls the most specialized interest adding method in the `BankAccount` class hierarchy on `ba`. The dynamic type of `ba` determines which `AddInterests` method to call. If, for instance, `ba` refers to a `LotteryAccount`, the `AddInterests` method of class `LotteryAccount` is used. Please notice that this is indeed the expected result:

> The type of the receiver object *obj* controls the interpretation of messages to *obj*.

And further, the most specialized method relative to the type of the receiver is called.

Let us - for a moment - assume that we do not have access to virtual methods and dynamic binding. In Program 28.27 we have rewritten Program 28.26 in such a way that we explicitly control the type dispatching. This is the part of Program 28.27 emphasized in **purple**. Thus, the **purple** parts of Program 28.26 and Program 28.27 are equivalent. Which version do you prefer? Imagine that many more bank account types were involved, and find out how valuable virtual methods can be for your future programs.

```
1  using System;
2
3  public class AccountClient{
4
5    public static void Main(){
6       BankAccount[] accounts =
7        new BankAccount[5]{
8          new CheckAccount("Per",1000.0M, 0.03),
9          new SavingsAccount("Poul",1000.0M, 0.03),
10         new CheckAccount("Kurt",1000.0M, 0.03),
11         new LotteryAccount("Bent",1000.0M),
12         new LotteryAccount("Lone",1000.0M)
```

```
13        };
14
15      foreach(BankAccount ba in accounts){
16        if (ba is CheckAccount)
17          ((CheckAccount)ba).AddInterests();
18        else if (ba is SavingsAccount)
19          ((SavingsAccount)ba).AddInterests();
20        else if (ba is LotteryAccount)
21          ((LotteryAccount)ba).AddInterests();
22        else if (ba is BankAccount)
23          ((BankAccount)ba).AddInterests();
24      }
25
26      foreach(BankAccount ba in accounts){
27        Console.WriteLine("{0}", ba);
28      }
29    }
30
31  }
```

Program 28.27 *Adding interests without use of dynamic binding - AddInterest is not virtual.*

Notice that for the purpose of Program 28.27 we have modified the bank account classes such that `AddInterests` is not virtual any more. Notice also, in line 22, that the last check of `ba` is against `BankAccount`. The check against `BankAccount` must be the last branch of the if-else chain because all the bank accounts `b` in the example satisfy the predicate `b is BankAccount`.

The outputs of Program 28.26 and Program 28.27 are identical, and they are shown in Listing 28.28. As it turns out, we were not lucky enough to get interests out of our lottery accounts.

```
1  Per's check account holds 1030,000 kroner
2  Poul's savings account holds 930,000 kroner
3  Kurt's check account holds 1030,000 kroner
4  Bent's lottery account holds 1000,0 kroner
5  Lone's lottery account holds 1000,0 kroner
```

Listing 28.28 *Output from the bank account programs.*

The use of virtual methods - and dynamic binding - covers a lot of type dispatching which in naive programs are expressed with **`if-else`** chains

28.16. Overriding the Equals method in a class

Lecture 7 - slide 37

The `Equals` instance method in class `Object` is a virtual method, see Section 28.3. The `Equals` method is intended to be redefined (overridden) in subclasses of class `Object`. The circumstances for redefining `Equals` have been discussed in Focus box 13.1.

It is tricky to do a correct overriding of the virtual **`Equals`** method in class **`Object`**

Below we summarize the issues involved when redefining `Equals` in one of our own classes.

- Cases to deal with when redefining the **`Equals`** method:
  - Comparison with `null` (*false*)
  - Comparison with an object of a different type (*false*)
  - Comparison with **`ReferenceEquals`** (*true*)
  - Comparison of fields in two objects of the same type
- Other rules when redefining **`Equals`**:
  - Must not lead to errors (no exceptions thrown)
  - The implemented equality should be *reflexive*, *symmetric* and *transitive*
- Additional work:
  - **`GetHashCode`** should also be redefined in accordance with **`Equals`**
    - If `o1.Equals(o2)` then `o1.GetHashCode() == o2.GetHashCode()`
  - If you overload the == operator
    - Also overload `!=`
    - Make sure that `o1 == o2` and `o1.Equals(o2)` return the same result

We illustrate the rules in Program 28.29, where we override the `Equals` method in class `BankAccount`.

```
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): this(owner, 0.0) {
15    }
16
17    public BankAccount(string owner, double interestRate) {
18       nextAccountNumber++;
19       accounts.Add(this);
20       this.accountNumber = nextAccountNumber;
21       this.interestRate = interestRate;
22       this.owner = owner;
23       this.balance = 0.0M;
24    }
25
26    public override bool Equals(Object obj){
27      if (obj == null)
28        return false;
29      else if (this.GetType() != obj.GetType())
30        return false;
31      else if (ReferenceEquals(this, obj))
32        return true;
33      else if (this.accountNumber == ((BankAccount)obj).accountNumber)
34        return true;
35      else return false;
36    }
37
38    public override int GetHashCode(){
39      return (int)accountNumber ^ (int)(accountNumber >> 32);
40      // XOR of low orders and high orders bits of accountNumber
41      // According to GetHashCode API recommendation.
```

```
42    }
43
44    /* Some methods are not included in this version */
45
46 }
```

Program 28.29 *Equals and GetHashCode Methods in class BankAccount.*

Please follow the pattern in Program 28.29 when you have to redefine `Equals` in your future classes.

28.17. Upcasting and downcasting in C#

Lecture 7 - slide 38

Upcasting and *downcasting* are common words in the literature about object-oriented programming. We have already used these words earlier in this material, see for instance Program 28.21.

Upcasting converts an object of a specialized type to a more general type

Downcasting converts an object from a general type to a more specialized type

Figure 28.3 *A specialization hierarchy of bank accounts*

Relative to Figure 28.3 we declare two `BankAccount` and two `LotteryAccount` variables in Program 28.30. After line 4 `ba2` refers to a `BankAccount` object, and `la2` refers to a `LotteryAccount` object.

The assignment in line 6 reflects an upcasting. `ba1` is allowed to refer to a `LotteryAccount`, because - conceptually - a `LotteryAccount` **is a** `BankAccount`.

In line 7, we attempt to assign `ba2` to `la1`. This is an attempted downcasting. This is statically invalid, and the compiler will always complain. Notice that in some cases the assignment `la1 = ba2` is legal, namely when `ba2` refers to a `LotteryAccount` object. In order to make the compiler happy, you should write `la1 = (LotteryAccount)ba2`.

In line 9 we attempt to do the downcasting discussed above, but it fails at run-time. The reason is - of course - that `ba2` refers to a `BankAccount` object, and not to a `LotteryAccount` object.

After having executed line 6, `ba1` refers to a `LotteryAccount` object. Thus, in line 11 we can assign `la1` to the reference in `ba1`. Again, this is a downcasting. As noticed above, the downcasting is necessary to calm the compiler.

```
    BankAccount    ba1,
                   ba2 =   new BankAccount("John", 250.0M, 0.01);
    LotteryAccount la1,
                   la2 =   new LotteryAccount("Bent", 100.0M);

    ba1 = la2;                      // upcasting   - OK
//  la1 = ba2;                      // downcasting - Illegal
                                    //   discovered at compile time
//  la1 = (LotteryAccount)ba2;      // downcasting - Illegal
                                    //   discovered at run time
    la1 = (LotteryAccount)ba1;      // downcasting - OK
                                    //   ba1 already refers to a LotteryAccount
```

Program 28.30 *An illustration of upcasting and downcasting.*

Upcasting and downcasting reflect different views on a given object

The object is not 'physically changed' due to upcasting or downcasting

The general rules of upcasting and downcasting in class hierarchies in C# can be expresses as follows:

- **Upcasting:**
  - Can occur implicitly during assignment and parameter passing
  - A natural consequence of polymorphism and the ***is-a*** relation
  - Can always take place
- **Downcasting:**
  - Must be done explicitly by use of type casting
  - Can not always take place

28.18. Inheritance and Variables

Lecture 7 - slide 40

We have focused a lot on methods in the previous sections. We will now summarize how variables are inherited.

Variables (fields) are inherited

Variables cannot be virtual

Variables are inherited. Thus a variable `v` in a superclass `A` is present in a subclass `B`. This is even the case if `v` is private in class `A`, see Exercise 7.2.

What happens if a variable `v` is present in both a superclass and a subclass? A variable can be redefined in the following sense:

- Same name in super- and subclass: two entirely different meanings (`new`)

We illustrate this situation in the ABC example of Program 28.31. Both class `A` and `B` have an `int` variable `v`. This can be called *accidental redefinition*, and this is handled in the program by marking `v` in class `B` with the modifier **`new`**.

Now, in the client class `App`, we make some `A` and `B` objects. In line 17-23 we see that the static type of a variable determines which version of `v` is accessed. Notice in particular the expression `anotherA.v`. If variable access had been virtual, `anotherA.v` would return the value 5. Now we need to adjust the static type explicitly with a type cast (see Section 28.12) to obtain a reference to `B.v`. This is illustrated in line 21.

```
1  using System;
2
3  public class A{
4    public int v = 1;
5  }
6
7  public class B: A{
8    public new int v = 5;
9  }
10
11 public class App{
12   public static void Main(){   // Static type    Dynamic type
13     A anA =      new A(),      //     A             A
14       anotherA = new B();      //     A             B
15     B aB =       new B();      //     B             B
16
17     Console.WriteLine(
18      "{0}",
19        anA.v               // 1
20        + anotherA.v        // 1
21        + ((B)anotherA).v   // 5
22        + aB.v              // 5
23     );
24   }
25 }
```

Program 28.31 *An illustration of "non-virtual variable access".*

We do not normally use public instance variables!

The idea of private instance variables and *representation independence* was discussed in Section 11.6.

28.19. References

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