Object-oriented Programming in C#

for C and Java programmers

February 2010

Kurt Nørmark ©

Department of Computer Science,

Aalborg University,

Denmark.

WEB version:

http://www.cs.aau.dk/~normark/oop-csharp/html/notes/theme-index.html

Abstract

This is a teaching material about object-oriented programming, illustrated with use of the programming language C#. The initial version was written i 2006.

It is assumed, as a prerequisite, that the readers have some knowledge about imperative programming, preferably knowledge about C. I believe that it is best to learn object-oriented programming after having learned and worked with imperative programming. Thus, we adhere to an "object later approach".

The starting point of of this teaching material is a number of slides. The slides are annotated with explanations and additional resources. The most comprehensive version of the material has the form of a traditional textbook. You can think of the textbook as grown on top of the slide material.

Almost all programs in this material have been developed together with the material. However, a few programs come from other sources. If the programs are not original, the source programmers are mentioned and acknowledged in an initial comment line of the program.

The current version of the material is complete up to (and including) the last chapter in lecture about Contracts (chapter 53). The final chapters - corresponding to the lectures about test and documentation - only contain slide material.

The teaching material is an online material represented in HTML. A PDF version of the textbook is also available. In order to limit the size of the PDF files some less important programs have been left out from the PDF edition. In the web edition (in HTML files) the full content is available.

We have used colors to emphasize aspects of the source programs in this material. It is therefore recommeded that you read the material from a color medium.

We would like to point out a couple of qualities of the web edition of the material. First, we have provided for extensive cross linking of the material. Whenever relevant, we have provided links from one part of the material to another. We have also provided a comprehensive subject index. Finally, there are separate indexes of source programs and exercises. The source program index provides links to complete, textual versions of the C# programs of the material, ready for you to compile and use.

In the spring of 2008 the material has been used in a course where the students have a background in imperative Visual Basic programming. We have therefore added a chapter that compares the fundamental (non-objected) parts of Visual Basic with the similar parts of C#. The chapter about Visual Basic is only available in the web-version.

Prior to the fall semester of 2008, selected aspects of C# 3.0 have been included in the material. This includes automatic properties, object initializers, extension methods, and lambda expressions.

The January 2009 edition includes a number of bug-fixes (as collected during the fall of 2008) and some new exercises targeted at the Aalborg University OOPA spring course 2009 at the BAIT education.

The February 2010 edition is a minor revision compared with the August 2009 edition. The February 2010 edition is the last, and final, version of the teaching material.

Kurt Nørmark
normark@cs.aau.dk
Department of Computer Science
Alborg University
Denmark
February 5, 2010

Colophon: This material has been made with the LENO system. LENO is an XML language which is defined by an XML DTD. LENO is used together with LAML. LAML is a software package that makes XML available in the Scheme programming language. For this version of the material, LAML version 35.0 (development version) has been used. This setup allows us to write LENO material in the programming language Scheme. Thus, the source of this material has been written in Scheme with use of the mirror functions of LENO XML language. Most illustrations have been written in an extension of SVG which allows for high-level representations of graphs. Batik 1.6 has been used to rasterize the SVG images in some versions of the material. The primary target format of the teaching material is HTML. LENO can be used to produce different views of the material. In the concrete material we make use of the slide view and the thematic view (the text book view). The aggregated slide view is also available. The PDF version of the thematic view (the text book view) has been made by PDF Creator (version 0.9.5). Text breaking and page enumeration has been made in Microsoft Word 2003 via 'Edit with Microsoft Word' in Internet Explorer version 7.0.

Contents

1. From structured programming to object-oriented programming 1
2. Towards Object-oriented Programming 7
3. Phenomena and Concepts 13
4. Towards Object-oriented Programs 19

5. The C# Language and System 23
6. C# in relation to C 25
7. C# in relation to Java 53
8. C# in relation to Visual Basic 57
9. C# Tools and IDEs 59

10. Classes: An Initial Example 61
11. Classes 67
12. Creating and Deleting Objects 89

13. Reference Types 97
14. Value Types 103
15. Organization of C# Programs 117
16. Patterns and Techniques 123

17. Accessing Data in Objects 133
18. Properties 135
19. Indexers 147
20. Methods 151

21. Overloaded Operators 165
22. Delegates 173
23. Events 183
24. Patterns and Techniques 189

25. Specialization of Classes 195
26. Extension of Classes 203
27. Inheritance in General 209
28. Inheritance in C# 215

29. Method Combination 241
30. Abstract Classes - Sealed Classes 245
31. Interfaces 253
32. Patterns and Techniques 271

33. Fundamental Questions about Exception Handling 301
34. Conventional Exception Handling 307
35. Object-oriented Exception Handling 311
36. Exceptions and Exception Handling in C# 313

37. Streams 333
38. Directories and Files 355
39. Serialization 359
40. Patterns and Techniques 369

41. Motivation for Generic Types 373
42. Generic Types 381
43. Generic Methods 401

44. Collections - History and Overview 407
45. Generic Collections in C# 409
46. Generic Dictionaries in C# 437
47. Non-generic Collections in C# 445
48. Patterns and Techniques 447

49. Correctness 453
50. Specification with preconditions and postconditions 457
51. Responsibilities and Contracts 463
52. Class Invariants 471

1. From structured programming to object-oriented programming

We will assume that the reader of this material has some knowledge of imperative programming, and that the reader already has been exposed to the ideas of structured programming. More specifically, we will assume that the reader has some background in C programming. In Chapter 6 (corresponding to the second lecture of the course) we summarize the relationships between C and C#.

1.1. Structured Programming

Lecture 1 - slide 2

We approach object-oriented programming by reviewing the dominating programming approach prior to object-oriented programming. It is called *structured programming*. A brief background on structured programming, imperative programming, and - more generally - different schools of programming is provided in Focus box 1.1. I will recommend that you read the Wikipedia article about structured programming [wiki-str-pro]. It captures, very nicely, the essence of the ideas.

Structured programming relies on use of high-level control structures instead of low-level jumping

Structured programming is also loosely coupled with *top-down programming* and *program development by stepwise refinement*

Structured programming covers several, loosely coupled ideas. As summarized above, one of these is the use of control structures (such as if, switch/case, while and for) instead of gotos.

Use of relatively small procedures is another idea. A well-structured program should devote a single procedure to the solution of a single problem. The splitting of problems in subproblems should be reflected by breaking down a single procedure into a number of procedures. The idea of *program development by stepwise refinement* [Wirth71] advocates that this is done in a top-down fashion. The items below summarize the way it is done.

- Start by writing the main program
  - Use selective and iterative control structures
  - Postulate and call procedures P1, ...,Pn
- Implement P1, ... Pn, and in turn the procedures they make use of
- Eventually, the procedures become so simple that they can be implemented without introducing additional procedures

Only few programmers are radical with respect to top-down structured programming. In the practical world it is probably much more typical to start somewhere in the middle, and then both work towards the top and towards the bottom.

Imperative programming, Structured programming, and Programming paradigms. FOCUS BOX 1.1

Imperative programming is one of the four main *programming paradigms*. The others are functional programming, object-oriented programming, and logic programming.

Imperative programming is closely related to the way low-level machine languages work: Commands are used to change the values of locations in the memory of the computer. In high-level languages, this is achieved by use of *assignment statements*, which is used to change the values of variables. The assignment statement is therefore the archetypical command in imperative programming. Control structures (sequence, selection, and iteration) come on top of that together with procedural abstractions.

Programming done in the early years of the computing era (before the introduction of Algol) is often thought of as "unstructured programming". Unstructured programming is largely characterized by use of "jumping around" by means of **goto** commands. The introduction of **if** and **while** control structures together with procedures eliminated the need for gotos. This can be shown theoretically, but - more important - it also holds true in the practical world of imperative programming. Armed with the common control structures (**if** and **while**, for instance) and procedural abstraction, very few programmers are tempted to use a *goto* statement in the programs they write. Such programming, without use of goto statements, is often called *structured programming*.

1.2. A structured program: Hangman

Lecture 1 - slide 3

In order to be concrete we will look at parts of a C program. The program implements a simple and rudimentary version of the well-known Hangman game. We will pretend that the program has been developed according to the structured programming ideas described in Section 1.1.

The main Hangman program, `main`, is shown in Program 1.1. The fragments shown in **purple** are postulated (in the sense discussed in Section 1.1). I.e., they are called, but not yet defined at the calling time. The postulated procedures are meant to be defined later in the program development process. Some of them are shown below.

```
int main(void){
  char *playerName;
  answer again;

  playerName = getPlayerName();
  initHangman();
  do{
    playHangman(playerName);
    again = askUser("Do you want to play again");
  } while (again == yes);
}
```

Program 1.1 *The main function of the Hangman program.*

The function `getPlayerName` is intended to prompt the Hangman player for his or her name. As it appears in Program 1.2 this function only uses functions from the C standard library. Therefore there are no emphasized parts in `getPlayerName`.

```
char *getPlayerName(){
  char *playerName = (char*)malloc(NAME_MAX);

  printf("What is your name? ");
  fgets(playerName, NAME_MAX, stdin);
  playerName[strlen(playerName)-1] = '\0';
  return playerName;
}
```

Program 1.2 *The function getPlayerName of main.*

The function `initHangman` calls an additional initialization function called `initPuzzles`, which reads a puzzle from a text file. We will here assume that this function does not give rise to additional refinement. We do not show the implementation of `initPuzzles`.

```
void initHangman (void){
  srand(time(NULL));
  initPuzzles("puzzles.txt");
}
```

Program 1.3 *The function initHangman of main.*

`askUser` is a general purpose function, which was called in `main` in Program 1.1. We show it in Program 1.4 (only on web) and we see that it does not rely on additional functions.

The function `playHangman`, seen in Program 1.5, is called by `main` in the outer loop in Program 1.1. `playHangman` contains an inner loop which is related to a single round of playing. As it appears from Program 1.5 `playHangman` calls a lot of additional functions (all emphasized, but not all of them included here).

```
void playHangman (char playerName[]){
  int aPuzzleNumber, wonGame;
  puzzle secretPuzzle;
  hangmanGameState gameState;
  char playersGuess;

  initGame(playerName, &gameState);
  aPuzzleNumber = rand() % numberOfPuzzles();
  secretPuzzle = getPuzzle(aPuzzleNumber);

  while ((gameState.numberOfWrongGuesses < N) &&
         (gameState.numberOfCorrectGuesses < secretPuzzle.numberOfCharsToGuess)){
    gameStatistics(gameState, secretPuzzle);
    presentPuzzleOutline(secretPuzzle,gameState);  printf("\n");
    presentRemainingAlphabet(gameState);  printf("\n");
    if (CHEATING) presentSecretPuzzle(secretPuzzle);
    printf("\n");
    playersGuess = getUsersGuess();
    clrconsole();
    updateGameState(&gameState, secretPuzzle, playersGuess);
  }
  gameStatistics(gameState, secretPuzzle);
  wonGame = wonOrLost(gameState,secretPuzzle);
  handleHighscore(gameState, secretPuzzle, wonGame);
}
```

Program 1.5 *The function playHangman of main.*

In Program 1.6 (only on web) and Program 1.7 (only on web), we show two additional functions, `initGame` and `getPuzzle`, both of which are called in `playHangman` in Program 1.5.

As already brought up in Section 1.1 many programmers do not strictly adhere to structured programming and top-down refinement when coding the hangman program. If *you* have programmed Hangman, or a similar game, it is an interesting exercise to reflect a little on the actual approach that was taken during your own development. In Section 4.1 we return to the Hangman example, restructured as an object-oriented program.

Exercise 1.1. *How did you program the Hangman game?*

This is an exercise for students who have a practical experience with the development of the Hangman program, or a similar game.

Recall how you carried out the development of the program.

To which degree did you adhere to *top-down development by stepwise refinement*?

If you did not use this development approach, then please try to characterize how you actually did it.

1.3. Observations about Structured Programming

Lecture 1 - slide 4

We will now attempt to summarize some of the weaknesses of structured programming. This will lead us towards object-oriented programming.

Structured programming is not *the wrong way* to write programs. Similarly, object-oriented programming is not necessarily *the right way*. Object-oriented programming (OOP) is an alternative program development technique that often tends to be better if we deal with large programs and if we care about program reusability.

We make the following observations about structured programming:

- Structured programming is narrowly oriented towards solving one particular problem
    - It would be nice if our programming efforts could be oriented more broadly
- Structured programming is carried out by gradual decomposition of the functionality
    - The structures formed by functionality/actions/control are *not* the most *stable* parts of a program
    - Focusing on data structures instead of control structure is an alternative approach
- Real systems have no single top - Real systems may have multiple tops [Bertrand Meyer]
    - It may therefore be natural to consider alternatives to the top-down approach

Let us briefly comment on each of the observations.

When we write a 'traditional' structured program it is most often the case that we have a single application in mind. This may also be the case when we write an object-oriented program. But with object-oriented programming it is more common - side by side with the development of the application - also to focus on development of program pieces that can be used and reused in different contexts.

The next observation deals with 'stable structures'. What is most stable: the overall control structure of the program, or the overall data structure of the program? The former relates to use of various control structures and to the flow procedure calls. The latter relates to data types and classes (in the sense to be discussed in Chapter 11). It is often argued that the overall program data structure changes less frequently than the overall program control structure. Therefore, it is probably better to base the program structure on decomposition of data types than on procedural decomposition.

The last observation is due to Bertrand Meyer [Meyer88]. He claims that "Real systems have no top". Let us take the Hangman program as an example. Even if it is likely that we can identify a single top of most hangman programs (in our program, `main` of Program 1.1) the major parts of the program should be able to survive in similar games, for instance in "Wheel of Fortune". In addition, a high score facility of Hangman should be applicable in a broad range of games. The high score part of the Hangman program may easily account for half of the total number of source lines in Hangman, and therefore it is attractive to reuse it in other similar games. The simple textual, line-oriented user interface could be replaceable by a more flexible user graphical user interface. In that way, even the simple Hangman program can easily be seen as a program with no top, or a program with multiple tops.

Readers interested in a good and extended discussion of 'the road to object-orientation' should read selected parts of Bertrand Meyers book 'Object-oriented Software Construction' [Meyer88]. The book illustrates object-oriented programming using the programming language Eiffel, and as such it is not directly applicable to the project of this course. The book is available in two versions. Either of them can be used. In my opinion 'Object-oriented Software Construction' is one of the best books about object-oriented programming.

1.4. Towards Object-oriented Programming

Lecture 1 - slide 5

We are now turning our interests towards 'the object-oriented way'. Below we list some of the most important ideas that we must care about when we make the transition from structured programming to object-oriented programming. This discussion is, in several ways, continued in Chapter 2.

- The gap between the problem and the level of the machine:
  - Fill the gap bottom up
- Use the data as the basic building blocks
  - Data, and relations between data, are *more stable* than the actions on data
- Bundle data with their natural operations
  - Build on the ideas of *abstract datatypes*
  - Consolidate the programming constructs that encapsulate data (structs/records)
- Concentrate on the *concepts and phenomena* which should be handled by the program
  - Make use of existing theories of phenomena and concepts
  - Form new concepts from existing concepts
- Make use of a programming style that allows us to collapse the programming of objects

Our approach to object-oriented programming is continued in Chapter 2. Before that, we will clarify the concept of abstract data types.

1.5. Abstract Datatypes

Lecture 1 - slide 10

A *data type* (or, for short, a *type*) is a set of values. All the values in a type share a number of properties. An *abstract data type* is a data type where we focus on the possible operations on the values in the type, in contrast to the representation of these values. This leads to the following definitions.

A *datatype* is a set of values with common properties. A datatype is a classification of data that reflects the intended use of the data in a program.

An *abstract datatype* is a data type together with a set of operations on the values of the type. The operations hide and protect the actual representation of the data.

In this material, boxes on a dark blue background with white letters are intended to give precise definitions of concepts.

To strengthen our understanding of abstract data types (ADTs) we will show a few *specifications* of well-known data types: Stacks, natural numbers, and booleans. A specification answers "what questions", not "how questions". The details are only shown in the web version of the material.

1.6. References

[Meyer88]	Bertrand Meyer, *Object-oriented software construction*. Prentice Hall, 1988.
[Wirth71]	Niklaus Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, Vol. 14, No. 4, April 1971, pp. 221-227.
[Wiki-str-pro]	Wikipedia: Structured_programming http://en.wikipedia.org/wiki/Structured_programming

2. Towards Object-oriented Programming

In this and the following chapter we will gradually unveil important theoretical and conceptual aspects of object-oriented programming. After this, in Chapter 4 we will be more concrete and practical, again in terms of the Hangman example.

In this chapter we will deal with a number of different aspects that lead in the direction of object-oriented programming. We do not attempt to relate these aspects to each other. Thus, in this chapter you will encounter a number of fragmented observations that - both individually and together - bring us towards object-oriented programming.

2.1. Client, Servers, and Messages

Lecture 1 - slide 7

We will start with message passing in between objects. One object (often called the *client*) sends a message to another object (often called the *server*). The client asks for a service. The server will do the job, and eventually return an answer to the client.

"Client" and "server" are general role names of objects. When the server receives a message, it may decide to forward the message to some subserver (because it cannot handle the request - solve the problem - itself). In this way, the server becomes a client of another server.

We will primarily be concerned with message passing where the client waits for an answer from the server. Thus, nothing happens in the client before the server has completed its work. This is referred to as *synchronous message passing*. *Asynchronous message passing* is also possible. This involves parallel activities. This is a slightly more advanced topic.

Peter orders a Pizza at AAU Pizza by email. Via interaction between a number of service providers, a pizza is delivered to Peters group room

Below we study an everyday example of message passing between an object (person) who orders a pizza, and a "Pizza server". The Pizza server relies on other subservers (subcontractors), in our example the butcher, the greengrocer, and a transport service. Thus, the Pizza crew are customers in other shops, and they make use of other services.

Notice that Peter - the hungry guy - is not aware of the subcontractors. Peter only cares about the interface of the Pizza server.

In some versions of this material you may interactively play the Pizza scenario in order to find out how the objects cooperate when Peter orders a Pizza. The scenario emphasizes that there is always a single current object (at least as long as we deal with synchronous message passing).

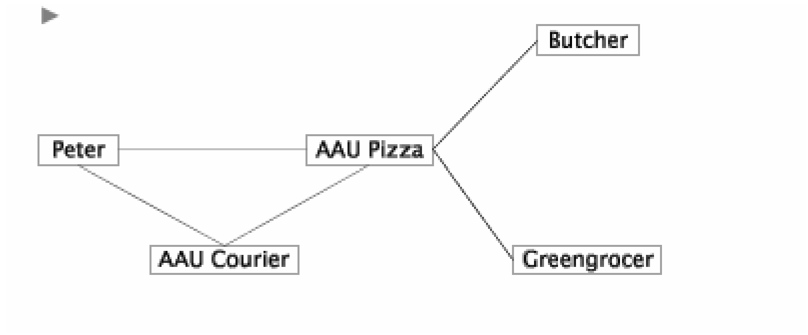

Figure 2.1 *The scenario of pizza ordering. The scenario focuses on a number of objects (persons) who communicate by message passing.*

Is it reasonable that Peter is idle in the period of time in between pizza ordering and pizza delivery? It depends on the circumstances. If you wait in the restaurant you may actually be left with feeling of 'just waiting'. If Peter orders the Pizza from his group room, Peter probably prefers to send an asynchronous message. In that way he can do some work before the pizza arrives. In this mode, we should, however, be able to handle the *interrupt* in terms of the actual pizza delivery. Again, this is a more advanced topic.

A ***client*** asks for a service at some given service provider (***server***).

This may lead the service provider (which now plays a client role) to ask for subservices

Clients and servers communicate by ***passing messages*** that return results

Try the accompanying SVG animation

In our model of message passing, it is inherent that messages return a result. Alternatively, we could use a model in which the 'the message result' is handled by a message in the other direction. We have chosen a model, which can be used directly in mainstream object-oriented programming languages (such as C#).

We will come back to clients and servers in the context of the lecture about classes, see Section 10.2. Message passing is taken up in that lecture, see Section 10.3.

2.2. Responsibilities

Lecture 1 - slide 8

Responsibility - and distribution of responsibility - is important in a network of cooperating objects. In Section 2.1 we studied a network of people and pizza makers. The Pizza maker has certain assumptions about orders from customers. We cannot expect the pizza maker to respond to an order where the customers want to buy a car, or a pet. On the other hand, the customer will be unhappy if he or she receives spaghetti (or a chocolate bar) after having ordered a pizza calzone.

Objects that act as servers manage a certain amount of responsibility

We will talk about the *responsibility of an object* as such. The object is responsible to keep the data, which it encapsulates, in good shape. It should not be possible to bring the object in an inconsistent state.

The *responsibility of an operation* of a class/object does also make good sense. If the sender of the message, which activates an operation fulfills certain (pre)conditions, it is the obligation of the operation to deliver a result which comply with a certain (post)condition.

The responsibilities of an object, together with the responsibilities of the operations of the object, sharpen the profile of the object, and they provide for a higher degree of cohesion of the object.

- Responsibility
    - Of an object, as reflected by the interface it provides to other objects
    - Of an operation
        - Precondition for activation - proposition about prerequisites for calling
        - Postcondition - proposition about result or effects
    - Well-defined responsibilities provide for coherent objects

In Chapter 49 through Chapter 53 we will devote an entire lecture to discussion of responsibilities, and how to specify the distribution of responsibilities among objects. This will involve *contracts*, which (again) is a real-world concept - a metaphor - from which can we can gain useful inspiration when we develop software.

You should care about the responsibilities of both objects and operations

The *distribution of responsibilities* will become a major theme later in the course

2.3. Data-centered modularity

Lecture 1 - slide 9

Message passing is mainly a dynamic (run-time) aspect of object-oriented programs. Let us now focus on a static aspect: modularity.

Modularity is the property of a computer program that measures the extent to which it has been composed out of separate parts called modules [Wikipedia]

Non-modular programs (programs written without decomposition) are unwieldy. The question we care about here is the kind of modularity to use together with abstract data types. We will identify the following kinds of modularity:

- **Procedural modularity**
    - Made up of individual procedures or functions
    - Relatively fine grained
    - Not sufficient for programming in the large
- **Boxing modularity**
    - A wall around arbitrary definitions
    - As coarse grained as needed
    - Visibility may be controlled - import and export
- **Data-centered modularity**
    - A module built around data that represents a single concept

- High degree of cohesion
- Visibility may be controlled
- The module may act as a datatype

Procedural modularity is used in structured programming, e.g. in C programs. It covers both functions and procedures. Procedural modularity remains to be very important, independent of programming paradigm!

Boxing modularity (our name of the concept) captures the module concept known from, e.g. Ada [Ada80] and Modula-2 [Wirth83]. In C, there are only few means to deal with boxing modularity. Most C programmers use the source files for boxing.

Boxing modularity allows us to box a data type and the operations that belong to the type in a module. When using data centered modularity *the module becomes a type itself*. This is an important observation. Object-oriented programming is based on data centered modularity.

Object-oriented programming is based on data-centered modularity

2.4. Reusability

Lecture 1 - slide 11

Let us now, for a moment, discuss reusability. The idea is that we wish to promote a programming style that allows us to use pieces of programs that we, or others, have already written, tested, and documented. Procedure libraries are well-known. Object-oriented programming brings us one step further, in the direction of class libraries. Class libraries can - to some degree - be thought of as reusable abstract data types.

More reuse - Less software to manage

We identity the following reusability challenges:

- Find
  - Where is the component, and how do I get it?
- Understand
  - What does the component offer, and how does it fit with my own program?
- Modify
  - Do I need to adapt the component in order to (re)use it?
- Integrate
  - How do I actually organize and use the component together with the existing components?

Finding has been eased a lot the last decade, due to the emergence of powerful search machines (servers!). Understanding is still a solid challenge. Documentation of reusable parts is important. Tools like JavaDoc (developed as on-line resources by Sun as part of the Java effort) are crucial. We will study interface documentation of class libraries later in this material. Modification should be used with great care. It is not a good idea to find a procedure or a class on the Internet, and rewrite it to fit your own needs. When the next version of the program is released you will be in great trouble. A modular modification approach, which separates your contributions from the original contributions, is needed. In object-oriented programming, inheritance alleviates this problem. The actual integration is relatively well-supported in modern object-oriented programming languages, because in these languages we have powerful means to deal with conflicts (such as name clashes) in between reused components and our own parts of the program.

2.5. Action on objects

Lecture 1 - slide 12

The final aspect that we want to bring up in our road towards object-oriented programming is the idea of action on objects. Actions should always be targeted at some object. Actions should not appear 'just up in the air'. Bertrand Meyer [Meyer88] has most likely been inspired by a famous John F. Kennedy quote when he formulated the idea in the following way:

Ask not what the system does: Ask what it does it to!
[Bertrand Meyer]

- Actions in general
  - Implemented by procedure calls
  - Often, but not always, with parameters
- Actions on objects
  - Activated via messages
    - A message always has a receiving object
    - A message is similar to a procedure calls with at least one actual parameter
  - A message activates an operation (a method)
    - The receiving object locates the best suited operation as responder (method lookup)

The activation of a concrete procedure or function is typically more complex than in ordinary imperative programming. The message is sent to an object. The reception of the message may cause the object to search for the best suited operation (method) to handle the request by the message. This process is sometimes called *method lookup*. In some object-oriented language the method lookup process is rather complicated.

In the next section we continue our road towards object-oriented programming, by discussing concepts and phenomena.

2.6. References

[Meyer88]	Bertrand Meyer, *Object-oriented software construction*. Prentice Hall, 1988.
[Wirth83]	Wirth, N., *Programming in Modula-2*, third. Springer-Verlag, 1985.
[Ada80]	*Ada Reference Manual*. United States Department of Defence, July 1980.

3. Phenomena and Concepts

Metaphors from the real life are important inspiration when we program the computer. It is limiting - and in fact counterproductive - to focus only on the technical computer concepts (bits, bytes, CPUs, memory words, USB ports, etc). According to my favorite dictionary (the American Heritage Dictionary of the English Language) a metaphor is

> "a figure of speech in which a word or phrase that ordinarily designates one thing is used to designate another, thus making an implicit comparison."

Many familiar programming concepts are not from the technical world of computers. Quite a few, such as int, float, and double come directly from mathematical counterparts. Messages and message passing, which we discussed in Section 2.1, are widely known from our everyday life. Even before email was invented, people communicated by means of messages (Morse codes, telegrams, postal mail letters).

It turns out that the ideas of classes and objects are - in part - inspired from the theory of concept and phenomena. We will unveil this in the following sections. Be warned that our coverage is brief and dense. It may very well be the case that it takes time for you to digest some of the ideas and concept that we are going to present.

3.1. Phenomena and Concepts

Lecture 1 - slide 14

A *phenomenon* is a thing that has definite, individual existence in reality or in the mind. Anything real in itself.

A *concept* is a generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection

The definitions of phenomenon and concept are taken from the PhD thesis of Jørgen Lindskov Knudsen and Kristine Stougaard Thomsen, Aarhus University [jlk-kst]. This thesis is also the source behind Section 3.2 - Section 3.4.

The characteristic aspects of a concept are the following:

- **The concept name**
- **The intension:** The collection of properties that characterize the phenomena in the extension of the concept
- **The extension:** The collection of phenomena that is covered by the concept

The name of the concept is also called the *designation*. The designation may cover a number of different names, under which the concept is known.

The word *intension* is used in the less well-known meaning (from logic) "the sum of the attributes contained in a term" (see for instance the American Heritage Dictionary of the English Language).

The word *extension* is used in the meaning (again from logic): "The class of objects designated by a specific term or concept" (according the same dictionary). Be careful not to confuse this meaning of *extension* with the more common meaning of the word, used for instance in Chapter 26 for extension of classes.

Concepts can be viewed in two different ways: The Aristotelian and the fuzzy way.

Using the *Aristotelian view*, the properties in the intension are divided into *defining properties* and *characteristic properties*. Each phenomenon of a concept must possess the defining properties. It is assumed that it is objectively determinable if a given phenomenon belongs to an Aristotelian concept.

Using the *fuzzy view*, the properties in the intension are only examples of possible properties. In addition to the example properties, the intension is also characterized by a set of prototypical phenomena. It is not objectively determinable if a given phenomenon belongs to a fuzzy concept.

We will primarily make use of the Aristotelian view on concepts. The relative sharp borderline between different concepts is attractive when we use concepts as the basis for the classes that we program in an object-oriented programming language. Many such classes represent *real-life concepts*, simply because many of our programs administrate things from the real world. It is, however, also common to make use of *imaginary concepts*, which have no real-life counterparts (such, as for instance, a hashtable).

3.2. Classification and exemplification

Lecture 1 - slide 15

To *classify* is to form a concept that covers a collection of similar phenomena.

To *exemplify* is to focus on a phenomenon in the extension of the concept

Classification and exemplification describe a relation between concepts and phenomena.

Classification forms a concept from a set of phenomena. The *intension* of the concept is the (defining) properties that are shared by the set of phenomena (according to the Aristotelian view).

Exemplification is the inverse of classification. Thus, the exemplification of a concept is a subset of the extension of the concept.

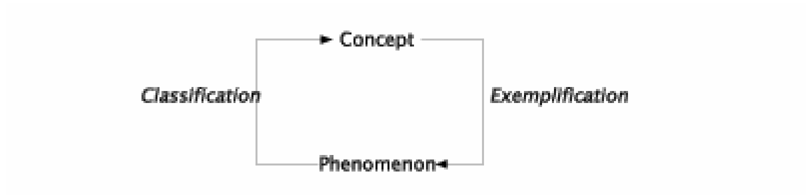

Figure 3.1 *The relationships between concepts and phenomena. Given a concept we can identify the examples of phenomena in the extension of the concept. Given such an example, we can (the other way around) find the concept that classifies the sample phenomenon.*

3.3. Aggregation and Decomposition

Lecture 1 - slide 16

In this and the following section we will see ways to form new concepts from existing concepts. First, we look at concepts related to 'parts' and 'wholes'.

To *aggregate* is to form a concept that covers a number of parts

To *decompose* is to split a concept into a number of parts

The concept of a house is an aggregation of (for instance) of the concepts wall, window, door, and roof. The latter are the decomposition of the house concept.

The intension of the aggregated concept corresponds to the intensions of the part concepts. But, in some cases, *the whole is more than the sum of its parts*. Thus, the aggregated concept may have additional properties as well.

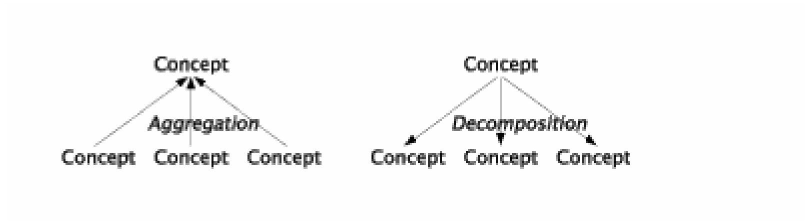

Figure 3.2 *An illustration of aggregation and decomposition. Notice that the relations between wholes and parts are in between concepts. Thus, aggregation and decomposition show how to form new concepts from existing concepts.*

In Figure 3.3 we show an example, namely the aggregation of a bike. Notice that we do not address the number of parts of the aggregated concept (no cardinalities). Following the tradition of UML notation, we use a diamond shape next to the aggregated concept. Notice, however, that it is not our intention to use exact UML notation in this material. We will primarily be concerned with programming notation, as defined (precisely) by a programming language.

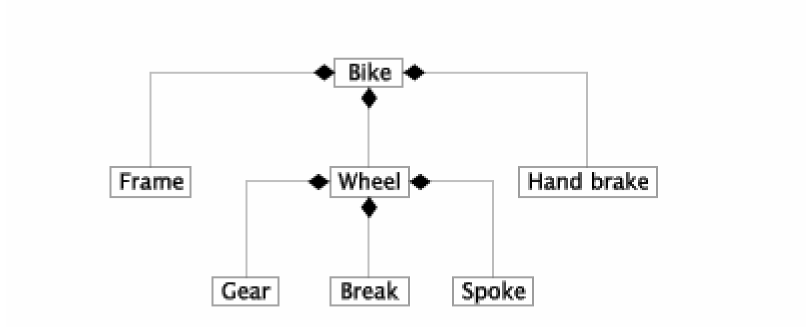

Figure 3.3 *An aggregation of a Bike in terms of Frame, Wheel, Brake, etc. This illustration does not capture the number of involved parts. Thus, the diagram does not capture the number of spokes per wheel, and the number of wheels per bike. The diamond shape is UML notation for aggregation.*

Exercise 1.2. *Aggregated Concepts*

Take a look at the concepts which are represented by the phenomena in the room where you are located. Identify at least four aggregated concepts. Enumerate the concepts of the decomposition.

3.4. Generalization and Specialization

Lecture 1 - slide 18

Generalization forms a broader concept from a narrow concept

Specialization forms a narrow concept from a broader concept

Generalization and specialization are seen as ways to form a new concept from an existing concept. The extension of a specialization S is a subset of the extension of the generalization G.

It is more difficult to capture specialization and generalization in terms of the intensions.

The concepts of Encyclopedia, Bible, and Dictionary are all specializations of the Book concept. Encyclopedia, Bibles and Dictionaries are all subsets of Books. It may be the case that the set of encyclopedia and the set of dictionaries are overlapping.

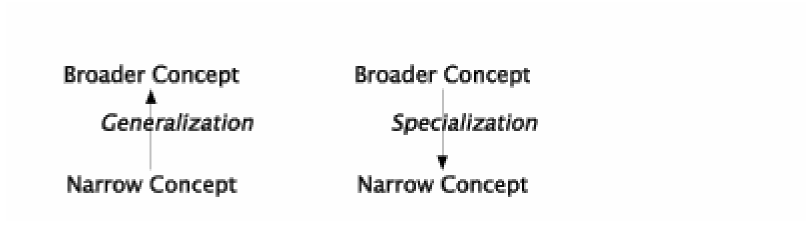

Figure 3.4 *An illustration of generalization and specialization.*

Below, in Figure 3.5 we show a generalization/specialization hierarchy of transportation concepts. Each parent in the tree is a generalization of its sons.

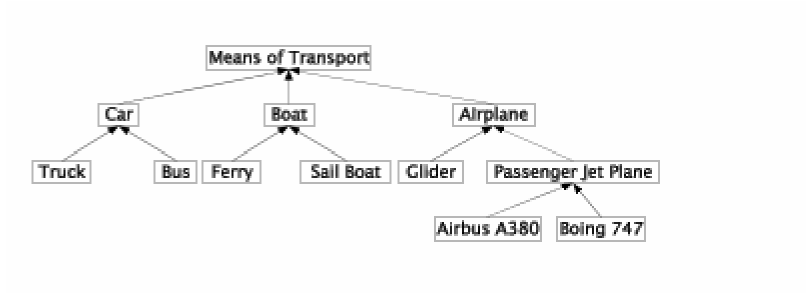

Figure 3.5 *A generalization/specialization hierarchy of 'Means of Transport'. All the concepts in this diagram are specialized means of transport. Notice that all the nodes in the specialization trees are concepts - not individual phenomena.*

The ideas of generalization and specialization among concepts are directly reflected in generalization and specialization among classes (see Chapter 25) as supported by inheritance in object-oriented programming languages.

Exercise 1.3. *Concepts and Phenomena*

The purpose of this exercise is to train your abilities to distinguish between concepts and phenomena.

Decide in each of the following cases if the mentioned item is a concept or a phenomena:

1. The door used to enter this room.
2. Todays issue of your favorite newspaper.
3. Your copy of today's issue of your favorite newspaper.
4. The collection of all copies of today's newpapers
5. Denmark.
6. European country.
7. The integer 7.
8. The set of integers between 1 and 10.
9. The set of all students who attend this course.
10. The oldest student who attend this course.

For an item considered as a phenomenon, identify the underlying concept.

Exercise 1.4. *University Concepts*

In a university study, the study activities are usually structured in a number of semesters. There are two kinds of study activitities: projects and courses. At Aalborg University, there are currently two kinds of courses: Study courses (dk: studieenhedskurser) and project courses (dk: projektenhedskurser).

Characterize the concepts of *university study*, *study activity*, *semester*, *project*, *course*, *study course*, and *project course* relative to Aggregation/Decomposition and Generalization/Specialization.

3.5. References

[Jlk-kst] A Conceptual Framework for Programming Languages: Jørgen Lindskov Knudsen and Kristine Stougaard Thomsen, Department of Computer Science, Aarhus Universitet, PB-192, April 1985.

4. Towards Object-oriented Programs

Below we will return to the example of the Hangman game, which we studied as a structured program in Section 1.2.

4.1. An object-oriented program: Hangman

Lecture 1 - slide 21

In Figure 4.1 we show a class diagram of our object-oriented version of the Hangman game.

The class `Puzzle` encapsulates the data of a single puzzle (the category and the word phrase). The class also offers an interface through which these informations can be accessed.

The class `PuzzleCollection` represents a number of puzzles. It is connected to the file system (or a database) where we keep the collection of puzzles while not playing a game. How this 'persistency' is actually handled is ignored for now.

Similar observations can be done for `HighscoreEntry` and `HighscoreList`.

The class `HangmanGame` encapsulates the state of a single (round of the) hangman game. It has associations to a player and a secret puzzle, and to the collections of puzzles and highscore entries. We do not in `HangmanGame` want to commit ourselves to any particular user interface. Thus, the actual user interface of the game is not part of the `HangmanGame` class.

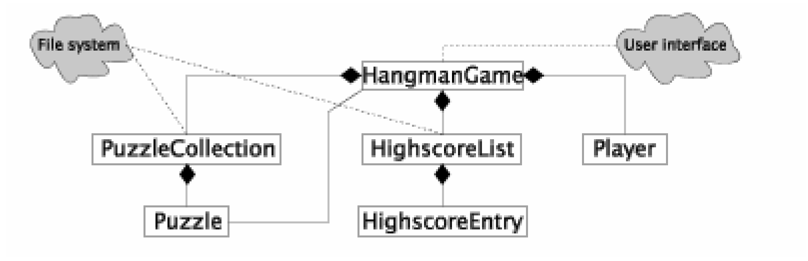

Figure 4.1 *The classes of a Hangman program. At the left hand side we see that PuzzleCollection is formed by Puzzle parts. Similarly, the HighscoreList is formed by HighScoreEntry parts. The HangManGame class if formed by three parts: PuzzleCollection, HighScoreList, and Player. Both file system and user interface aspects are "cloudy" in this diagram.*

Below we show sketches of the individual classes in the game. The classes and all the operations are marked as abstract, because the operations of the classes are not implemented, and because the current OOP version of the Hangman game is written at a very high level of abstraction. The concatenation of all classes in Program 4.1 - Program 4.5 can actually be compiled with a C# compiler. Abstract classes are discussed in Chapter 30.

The operation interfaces of the classes are most probably not yet complete, but they are complete enough to let you have an impression of the object-oriented programming approach.

```
abstract class Puzzle {

  public abstract string Category{
    get;
  }

  public abstract string PuzzlePhrase{
    get;
  }

  public abstract int NumberOfCharsToGuess();
}
```

Program 4.1 *The class Puzzle.*

Given that Program 4.1 - Program 4.6 contain so many abstract operations we will touch a little bit on what this means. It is not intended that you should learn the details of abstract classes here, however. This is the topic of Section 30.1. As noticed above, the abstract class shown in Program 4.1 can actually be compiled with a C# compiler. But it is clear that the class cannot be instantiated (no objects can be made). It is necessary to create a subclass of `Puzzle` in which we give the details of the abstract operations (methods and properties). Subclassing and inheritance will be discussed in Chapter 25 and subsequent chapters. In the subclass of `Puzzle` we need to supply puzzle data representation details.

```
abstract class HighScoreEntry {

  public abstract Player Player {
    get;
  }

  public abstract int Score{
    get;
  }
}
```

Program 4.2 *The class HighScoreEntry.*

Let us make a technical remark related to programming of abstract classes in C#. It is necessary to mark all operations (methods and properties) without bodies as 'abstract'. When a class contains at least one abstract operation, the class itself must also be marked as abstract. It is not sufficient to use the abstract modifier on the class.

```
abstract class Player {

  public abstract string Name{
    get;
  }

}
```

Program 4.3 *The class Player.*

```
abstract class PuzzleCollection {

  public abstract string Count{
    get;
  }

  public abstract Puzzle this[int i]{
    get;
  }

  public abstract void Add(Puzzle p);

}
```

Program 4.4 *The class PuzzleCollection.*

```
abstract class HighScoreList {

  /* Invariant: Entries always sorted */

  public abstract void Open(string FileName);

  public abstract string Count{
    get;
  }

  public abstract HighScoreEntry this[int i]{
    get;
  }

  public abstract void Add(HighScoreEntry e);

  public abstract void Close();

}
```

Program 4.5 *The class HighScoreList.*

The class `HangmanGame` in Program 4.6 (only on web) shows an outline of top-level class, cf. Figure 4.1. The operations in this class are intended to be called directly or indirectly by the `Main` method (not shown).

5. The C# Language and System

This chapter, together with Chapter 6, Chapter 7, and Chapter 9, is an introduction to the C# language and the C# system. On Windows, the latter is known as .Net. On purpose, we will keep the .Net part of the material very short. Our main interest in this lecture is how to program in C#, and how this is related to programming in other languages such as C, Java, and Visual Basic.

5.1. C# seen in a historic perspective

Lecture 2 - slide 2

It is important to realize that C# stands on the shoulders of other similar object-oriented programming languages. Most notably, C# is heavily inspired by Java. Java, in turn, is inspired by C++, which again - on the object-oriented side - can be traced back to Simula (and, of course, to C on the imperative side).

Here is an overview of the most important object-oriented programming languages from which C# has been derived:

- Simula (1967)
  - The very first object-oriented programming language
- C++ (1983)
  - The first object-oriented programming language in the C family of languages
- Java (1995)
  - Sun's object-oriented programming language
- C# (2001)
  - Microsoft's object-oriented programming language

5.2. The Common Language Infrastructure

Lecture 2 - slide 3

The Common Language Infrastructure (CLI) is a specification that allows several different programming languages to be used together on a given platform. The CLI has a lot of components, typically referred to by three-letter abbreviations (acronyms). Here are the most important parts of the Common Language Infrastructure:

- Common Intermediate language (CIL) including a common type system (CTS)
- Common Language Specification (CLS) - shared by all languages
- Virtual Execution System (VES)
- Metadata about types, dependent libraries, attributes, and more

The following illustration, taken from Wikipedia, illustrates the CLI and its context.

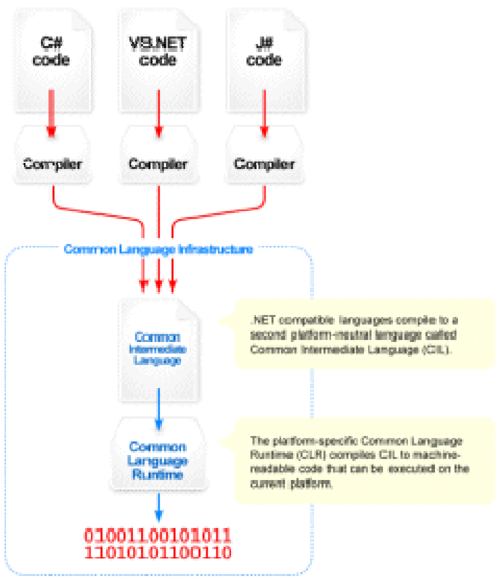
C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

Figure 5.1   *Wikipedia's overview diagram of the CLI*

.Net is one particular implementation of the Common Language Infrastructure, and it is undoubtedly the most complete one. .Net is closely associated with Windows. .Net is, however, not the only implementation of the CLI. Mono is another one, which is intended to work on several platforms. Mono is the primary implementation of the CLI on Linux. Mono is also available on Windows.

MONO and .NET are both implementations of the Common Language Infrastructure

The C# language and the Common Language Infrastructure are standardized by ECMA and ISO

5.3. C# Compilation and Execution

Lecture 2 - slide 5

The Common Language Infrastructure supports a two-step compilation process

- Compilation
  - The C# compiler: Translation of C# source to CIL
  - Produces **`.dll`** and **`.exe`** files
  - *Just in time* compilation: Translation of CIL to machine code
- Execution
  - With interleaved *Just in Time* compilation
  - On Mono: Explicit activation of the interpreter
  - On Window: Transparent activation of the interpreter

`.dll` and **`.exe`** files are - with some limitations - portable in between different platforms

6. C# in relation to C

As already mentioned in Chapter 1, this material is primarily targeted at people who know the C programming language. With this outset, we do not need to dwell on issues such as elementary types, operators, and control structures. The reason is that C and C# (and Java for that sake) are similar in these respects.

In this chapter we will discuss the aspects of C# which have obvious counterparts in C. Hopefully, the chapter will be helpful to C programmers who have an interest in C# programming.

In this chapter 'C#' refers to C# version 2.0. When we discuss C we refer to ANSI C ala 1989.

6.1. Simple types

Lecture 2 - slide 7

C supports the simple types `char`, `bool`, `int`, `float` and `double`. In addition there are a number of variation of some of these. In this context, we will also consider pointers as simple types.

The major differences between C# and C with respect to simple types are the following:

- All simple C# types have fixed bit sizes
- C# has a boolean type called **`bool`**
- C# chars are 16 bit long
- In C# there is a high-precision 128 bit numeric fixed-point type called **`decimal`**
- Pointers are not supported in the normal parts of a C# program
    - In the unsafe part C# allows for pointers like in C
- All simple types are in reality structs in C#, and therefore they have members

In C it is not possible to tell the bit sizes of the simple types. In some C implementations an `int`, for instance, will made by 4 bytes (32 bits), but in other C implementations an `int` may be longer or shorter. In C# (as well as in Java) the bit sizes of the simple types are defined and fixed as part of the specification of the language.

In C there is no boolean type. Boolean false is represented by zero values (such as integer 0) and boolean true is represented by any other integer value (such as the integer 1). In C# there is a boolean type, named `bool`, that contain the natural values denoted by `true` and `false`.

The handling of characters is messy in C. Characters in C are supported by the type named `char`. The `char` type is an integer type. There is a great deal of confusion about signed and unsigned characters. Typically, characters are represented by 8 bits in C, allowing for representation of the extended ASCII alphabet. In C# the type `char` represent 16 bits characters. In many respects, the C# type `char` corresponds to the Unicode alphabet. However, 16 bits are not sufficient for representation of all characters in the Unicode alphabet. The issue of character representation, for instance in text files, relative to the type `char` is a complex issue in C#. In this material it will be discussed in the lecture about IO, starting in Chapter 37. More specifically, you should consult Section 37.7.

The high-precision, 128 bit type called `decimal` is new in C#. It is a decimal floating point type (as opposed to `float` and `double` which are binary floating point types). Values in type `decimal` are intended for financial calculations. Internally, a decimal value consists of a sign bit (representing positive or negative), a 96 bit integer (mantissa) and a scaling factor (exponent) implicitly between 10^0 and 10^{-28}. The 96 bit integer part allows for representation of (at least) 28 decimal digits. The decimal exponent allows you to set the decimal point anywhere in the 28 decimal number. The decimal type uses 3 * 4 = 12 bytes for the mantissa and 4 bytes for the exponent. (Not all bits in the exponent are used, however). For more information, see [decimal-floating-point].

C pointers are not intended to be used in C#. However, C pointers are actually supported in the part of C# known as the unsafe part of the language. The concept of references is very important in C#. References and pointers are similar, but there are several differences as well. Pointers and references will be contrasted and discussed below, in Section 6.5.

All simple types in C# are in reality represented as structs (but not all structs are simple types). As such, this classifies the simple types in C# as *value types*, as a contrast to *reference types*. In addition, in C#, this provides for definition of methods of simple types. Structs are discussed in Section 6.6.

Below we show concrete C# program fragments which demonstrate some aspects of simple types.

```
using System;

class BoolDemo{

  public static void Main(){
    bool b1, b2;
    b1 = true; b2 = default(bool);
    Console.WriteLine("The value of b2 is {0}", b2);  // False
  }

}
```

Program 6.1 *Demonstrations of the simple type bool in C#.*

In Program 6.1 we have emphasized the parts that relate to the type `bool`. We declare two boolean variables `b1` and `b2`, and we initialize them in the line below their declarations. Notice the possibility of asking for the default value of type `bool`. This possibility also applies to other types. The output of Program 6.1 reveals that the default value of type `bool` is false.

```
using System;

class CharDemo{

  public static void Main(){
    char ch1 = 'A',
         ch2 = '\u0041',
         ch3 = '\u00c6', ch4 = '\u00d8', ch5 = '\u00c5',
         ch6;

    Console.WriteLine("ch1 is a letter: {0}", char.IsLetter(ch1));

    Console.WriteLine("{0} {1} {2}", ch3, ch4, char.ToLower(ch5));

    ch6 = char.Parse("B");
    Console.WriteLine("{0} {1}", char.GetNumericValue('3'),
                                 char.GetNumericValue('a'));
  }

}
```

Program 6.2 *Demonstrations of the simple type char in C#.*

In Program 6.2 we demonstrate the C# type `char`. We declare a number of variables, `ch1` ... `ch6`, of type `char`. `ch1` ... `ch5` are immediately initialized. Notice the use of single quote character notation, such as `'A'`. This is similar to the notation used in C. Also notice the **`'\u....'`** *escape notation*. This is four digit unicode character notation. Each dot in **`'\u....'`** must be a hexadecimal digit between 0 and f (15). The unicode notation can be used to denote characters, which are not necessarily available on your keyboard, such as the Danish letters Æ, Ø and Å shown in Program 6.2. Notice also the `char` operations, such as `char.IsLetter`, which is applied on `ch1` in the program. Technically, `IsLetter` is a static method in the struct `Char` (see Section 6.6 and Section 14.3 for an introduction to structs). There are many similar operations that classify characters. These operations correspond to the abstractions (macros) in the C library `ctype.h`. It is recommended that you - as an exercise - locate `IsLetter` in the C# library documentation. It is important that you are able to find information about already existing types in the documentation pages. See also Exercise 2.1.

Number Systems and Hexadecimal Numbers

FOCUS BOX 6.1

In the program that demonstrated the type char we have seen examples of hexadecimal numbers. It is worthwhile to understand why hexadecimal numbers are used for these purposes. This side box is a crash course on number systems and hexadecimal numbers.

The normal numbers are decimal, using base 10. The meaning of the number 123 is

$$1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

The important observation is that we can use an arbitrary base *b*, *b* > 1 as an alternative to 10 for decomposition of a number. Base numbers which are powers of 2 are particularly useful. If we use base 2 we get the binary numbers. Binary numbers correspond directly to the raw digital representation used in computers. The binary notation of 123 is 1111011 because

$1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$

is equal to the decimal number 123.

Binary numbers are important when we approach the lower level of a computer, but as can be seen above, binary numbers are unwieldy and not very practical. Hexadecimal numbers are used instead. Hexadecimal numbers use base 16, which is 2^4. We need 16 digits for notation of hexadecimal numbers. The first 10 digits are 0 .. 9. In lack of better notation, we use the letters A .. F as the six last digits. A = 10, ..., F = 15.

The important observation is that *a group of four binary digits (corresponding to four bits) can be translated to a single hexadecimal number*. Thus, we can immediately translate the binary number 01111011 to the two hexadecimal digits 7 and 11. These two hexadecimal digits are denoted as 7 and B respectively. With this observation, at single byte of eight bits can written as exactly two hexadecimal digits. Grouping the bits of 1111011 leads to 0111 1011. 0111 is 7. 1011 is 11 which is denoted by the hexadecimal digit B. The hexadecimal number 7B means

$7 * 16^1 + 11 * 16^0$

which is 123 (in decimal notation).

The explantion above is - in a nutshell - the reason why you should care about hexadecimal numbers. In Exercise 2.2 we will write programs that deal with hexadecimal numbers.

```
using System;
using System.Globalization;

class NumberDemo{

  public static void Main(){
    sbyte        sb1 = sbyte.MinValue;       // Signed 8 bit integer
    System.SByte sb2 = System.SByte.MaxValue;
    Console.WriteLine("sbyte: {0} : {1}", sb1, sb2);

    byte        b1 = byte.MinValue;          // Unsigned 8 bit integer
    System.Byte b2 = System.Byte.MaxValue;
    Console.WriteLine("byte: {0} : {1}", b1, b2);

    short        s1 = short.MinValue;        // Signed 16 bit integer
    System.Int16 s2 = System.Int16.MaxValue;
    Console.WriteLine("short: {0} : {1}", s1, s2);

    ushort         us1 = ushort.MinValue;    // Unsigned 16 bit integer
    System.UInt16  us2= System.UInt16.MaxValue;
    Console.WriteLine("ushort: {0} : {1}", us1, us2);

    int          i1 = int.MinValue;          // Signed 32 bit integer
    System.Int32 i2 = System.Int32.MaxValue;
    Console.WriteLine("int: {0} : {1}", i1, i2);

    uint           ui1 = uint.MinValue;      // Unsigned 32 bit integer
    System.UInt32  ui2= System.UInt32.MaxValue;
    Console.WriteLine("uint: {0} : {1}", ui1, ui2);

    long         l1 = long.MinValue;         // Signed 64 bit integer
    System.Int64 l2 = System.Int64.MaxValue;
    Console.WriteLine("long: {0} : {1}", l1, l2);

    ulong           ul1 = ulong.MinValue;    // Unsigned 64 bit integer
    System.UInt64   ul2= System.UInt64.MaxValue;
    Console.WriteLine("ulong: {0} : {1}", ul1, ul2);

    float           f1 = float.MinValue;     // 32 bit floating-point
    System.Single   f2= System.Single.MaxValue;
    Console.WriteLine("float: {0} : {1}", f1, f2);

    double          d1 = double.MinValue;    // 64 bit floating-point
    System.Double   d2= System.Double.MaxValue;
    Console.WriteLine("double: {0} : {1}", d1, d2);

    decimal         dm1 = decimal.MinValue;  // 128 bit fixed-point
    System.Decimal  dm2= System.Decimal.MaxValue;
    Console.WriteLine("decimal: {0} : {1}", dm1, dm2);


    string s = sb1.ToString(),
           t = 123.ToString();

  }


}
```

Program 6.3 *Demonstrations of numeric types in C#.*

In Program 6.3 we show a program that demonstrates all numeric types in C#. For illustrative purposes, we use both the simple type names (such as `int`, shown in **purple**) and the underlying struct type names (such as `System.Int32` shown in **blue**). To give you a feeling of the ranges of the types, the program prints the smallest and the largest value for each numeric type. At the bottom of Program 6.3 we show how the operation `ToString` can be used for conversion from a numeric type to the type `string`. The output of the numeric demo program is shown in Listing 6.4 (only on web).

Hexadecimal Numbers in C# **FOCUS BOX 6.2**

In Focus box 6.1 we studied hexadecimal numbers. We will now see how to deal with hexadecimal numbers in C#.

A number prefixed with 0x is written in hexadecimal notation. Thus, 0x123 is equal to the decimal number 291.

In C and Java the prefix 0 is used for octal notation. Thus, in C and Java 0123 is equal to the decimal number 83. This convention is not used in C#. In C#, 0123 is just a decimal number prefixed with a redundant digit 0.

While *prefixes* are used for encoding of number systems, *suffixes* of number constants are used for encoding of numerical types. As an example, 0X123L denotes a hexadecimal constant of type `long` (a 64 bit integer). The following suffixes can be used for integer types: **U** (unsigned), **L** (long), and **UL** (unsigned long). The following suffixes can be used for real types: **F** (float), **D** (double), and **M** (decimal). Both lowercase and uppercase suffixes will work.

A number can formatted in both decimal and hexadecimal notation. In the context of a `Console.WriteLine` call, the format specification (or placeholder) `{i:X}` will write the value of the variable `i` in hexadecimal notation. This is demonstrated by the following C# program:

```
using System;
class NumberDemo{
  public static void Main(){
     int i = 0123,
         j = 291;
     long k = 0X123L;

     Console.WriteLine("{0:X}", i);   // 7B
     Console.WriteLine("{0:D}", i);   // 123
     Console.WriteLine("{0:X}", j);   // 123
     Console.WriteLine("{0:D}", k);   // 291
  }
}
```

In the program shown above, **D** means decimal and **X** means hexadecimal. Some additional formattings are also provided for numbers: **C** (currency notation), **E** (exponential notation), **F** (fixed point notation), **G** (Compact general notation), **N** (number notation), **P** (percent notation), **R** (round trip notation for float and double). You should consult the online documentation for additional explanations.

Exercise 2.1. *Exploring the type Char*

The type `System.Char` (a struct) contains a number of useful methods, and a couple of constants.

Locate the type `System.Char` in your C# documentation and take a look at the methods available on characters.

You may ask where you find the C# documentation. There are several possibilities. You can find it at the Microsoft MSDN web site at `msdn.microsoft.com`. It is also integrated in Visual Studio and - to some degree - in Visual C# express. It comes with the C# SDK, as a separate browser. It is also part of the documentation web pages that comes with Mono. If you are a Windows user I will recommend the Windows SDK Documentation Browser which is bundled with the C# SDK.

Along the line of the character demo program above, write a small C# program that uses the `char` predicates `IsDigit`, `IsPunctuation`, and `IsSeparator`.

It may be useful to find the code position - also known as the *code point* - of a character. As an example, the code position of 'A' is 65. Is there a method in `System.Char` which gives access to this information? If not, can you find another way to find the code position of a character?

Be sure to understand the semantics (meaning) of the method `GetNumericValue` in type `Char`.

Exercise 2.2. *Hexadecimal numbers*

In this exercise we will write a program that can convert between decimal and hexadecimal notation of numbers. Please consult the focus boxes about hexadecimal numbers in the text book version if you need to.

You might expect that this functionality is already present in the C# libraries. And to some degree, it is.

The static method `ToInt32(string, Int32)` in class `Convert` converts the string representation of a number (the first parameter) to an arbitrary number system (the second parameter). Similar methods exist for other integer types.

The method `ToString(string)` in the struct `Int32`, can be used for conversion from an integer to a hexadecimal number, represented as a string. The parameter of `ToString` is a format string. If you pass the string "X" you get a hexadecimal number.

The program below shows examples:

```
using System;
class NumberDemo{
  public static void Main(){
     int i = Convert.ToInt32("7B", 16);     // hexadecimal 7B (in base 16) ->
                                            // decimal 123
     Console.WriteLine(i);                  // 123

     Console.WriteLine(123.ToString("X"));  // decimal 123 -> hexadecimal 7B
  }
}
```

Now, write a method which converts a list (or array) of digits in base 16 (or more generally, base *b*, *b* >= 2) to a decimal number.

The other way around, write a method which converts a positive decimal integer to a list (or array) of digits in base 16 (or more generally, base *b*).

Here is an example where the requested methods are used:

```
    public static void Main(){
      int r = BaseBToDecimal(16, new List{7, 11});  // 7B  -> 123
      List s = DecimalToBaseB(16, 123);             // 123 -> {7, 11} = 7B
      List t = DecimalToBaseB(2, 123);              // 123 -> {1, 1, 1, 1, 0, 1, 1 } =
                                                         // 1111011
      Console.WriteLine(r);
      foreach (int digit in s) Console.Write("{0} ", digit);  Console.WriteLine();
      foreach (int digit in t) Console.Write("{0} ", digit);
    }
```

6.2. Enumerations types

Lecture 2 - slide 8

Enumeration types in C# are similar to enumeration types in C, but a number of extensions have been introduced in C#:

- Enumeration types of several different *underlying types* can be defined (not just `int`)
- Enumeration types inherit a number of methods from the type **`System.Enum`**
- The symbolic enumeration constants can be printed (not just the underlying number)
- Values, for which no enumeration constant exist, can be dealt with
- Combined enumerations represent a collection of enumerations

Below, in Program 6.5 we see that the enumeration type `OnOff` is based on the type `byte`. The enumeration type `Ranking` is - per default - based on `int`.

```
1  using System;
2  
3  class NonSimpleTypeDemo{
4  
5    public enum Ranking {Bad, OK, Good}
6  
7    public enum OnOff: byte{
8      On = 1, Off = 0}
9  
10   public static void Main(){
11     OnOff status = OnOff.On;
12     Console.WriteLine();
13     Console.WriteLine("Status is {0}", status);
14 
15     Ranking r = Ranking.OK;
16     Console.WriteLine("Ranking is {0}", r  );
17     Console.WriteLine("Ranking is {0}", r+1);
18     Console.WriteLine("Ranking is {0}", r+2);
19 
20     bool res1 = Enum.IsDefined(typeof(Ranking), 3);
21     Console.WriteLine("{0} defined: {1}", 3, res1);
22 
23     bool res2= Enum.IsDefined(typeof(Ranking), Ranking.Good);
24     Console.WriteLine("{0} defined: {1}", Ranking.Good , res2);
25 
26     bool res3= Enum.IsDefined(typeof(Ranking), 2);
27     Console.WriteLine("{0} defined: {1}", 2 , res3);
28 
29     foreach(string s in Enum.GetNames(typeof(Ranking)))
30        Console.WriteLine(s);
31   }
32 
33 }
```

Program 6.5 *Demonstration of enumeration types in C#.*

In the example the methods `IsDefined` and `GetNames` are examples of static methods inherited from `System.Enum`.

In line 13 of Program 6.5 `On` is printed. In a similar C program, the number 1 would be printed.

In line 16 OK is printed, and line 17 prints Good. In line 18 the value of `r` + 2 is 3, which does not correspond to any of the values in type `Ranking`. Therefore the base value 3 is printed.

All the output of Program 6.5 is listed in Listing 6.6.

Combined enumeration (sometimes known as *flags enumeration*) is a slightly more advanced concept. We introduce it in Focus box 6.3.

Let us point out some additional interesting details in Program 6.5. There are two enumeration types in the program, namely a type called `Ranking` and another called `OnOff`. When we declare variables, the types `Ranking` and `OnOff` are used via their names. C programmer will be relieved to find out that it is not necessary (and not possible) to write `enum Ranking` and `enum OnOff`. Thus, no C-like `typedefs` are necessary to introduce simple naming.

In order to disambiguate the referencing of constants in an enumeration type, dot notation ala `Ranking.OK` must always be used. In the same way as in C, the enumeration constants have associated an integer value. The operation `IsDefined` allows us to check if a given value belongs to an enumeration type. `IsDefined` is an operation (a method) in a struct called `Enum`.

As a very pleasant surprise for the C programmer, there is access to the names of enumeration constants from the program. We show in the program that the expressions `Enum.GetNames(typeof(Ranking))` returns a string array with the elements "Bad", "OK", and "Good". In the same direction - as we have already seen above - it is possible to print the symbolic names of the enumeration constants. This is very useful. In C programs we always need a tedious `switch` to obtain a similar effect..

Combined Enumerations **FOCUS BOX 6.3**

Combined enumerations can be used to deal with small sets symbolic constants. Here is an example:

```
  [Flags]
  public enum Direction: byte{
    North = 1, East = 2, South = 4, West = 8,
  }
```

The first thing to notice is the mapping of symbolic constants to powers of two. We can form an expression `North | West` which is the bitwise combination of the underlying integer values 1 and 8. **|** is a bitwise or operator, see Table 6.1. At the binary level, 1 | 8 is equivalent to 0001 | 1000 = 1001, which represents the number 9. You should think of 1001 as a bit array where the leftmost bit is the encoding of `West` and the rightmost bit is the encoding of `North`.

`[Flags]` is an application of an attribute, see Section 39.6. It instructs the compiler to generate symbolic names of combinations, such as the composite name `North, West` in the example below.

We can program with the enumeration type in the following way:

```
    Direction d = Direction.North | Direction.West;
    Console.WriteLine("Direction {0}", d);                // Direction North, West
    Console.WriteLine("Direction {0}", (int)d);           // 9
    Console.WriteLine(HasDirection(d, Direction.North));  // True
    Console.WriteLine(HasDirection(d, Direction.South));  // False
```

The method `HasDirection` is a method we have programmed in the following way:

```
  // Is d in the direction e
  public static bool HasDirection(Direction d, Direction e){
    return (d & e) == e;
  }
```

It checks if `e` is contained in `d` in the a bitwise sense. It is also possible to name some of the combinations explicitly in the enumeration type:

```
  [Flags]
  public enum Direction: byte{
    North = 1, East = 2, South = 4, West = 8,
    NorthWest = North | West, NorthEast = North | East,
    SouthWest = South | West, SouthEast = South | East
  }
```

Exercise 2.3. *ECTS Grades*

Define an enumeration type `ECTSGrade` of the grades A, B, C, D, E, Fx and F and associate the Danish 7-step grades 12, 10, 7, 4, 2, 0, and -3 to the symbolic ECTS grades.

What is the most natural *underlying type* of `ECTSGrade?`

Write a small program which illustrates how to use the new enumeration type.

Exercise 2.4. *Use of Enumeration types*

Consult the documentation of type type `System.Enum`, and get a general overview of the methods in this struct.

Be sure that you are able to find the documentation of `System.Enum`

Test drive the example `EnumTest`, which is part of MicroSoft's documentation. Be sure to understand the program relative to its output.

Write your own program with a simple enumeration type. Use the `Enum.CompareTo` method to compare two of the values of your enumeration type.

6.3. Non-simple types

Lecture 2 - slide 9

The most important non-simple types are defined by classes and structs. These types define non-atomic objects/values. Because C# is a very rich language, there are other non-simple types as well, such as interfaces and delegates. We will not discuss these in this chapter, but they will play important roles in later chapters.

The most important similarities between C and C# with respect to non-simple types can be summarized in the following way:

- Arrays in C#: Indexed from 0. Jagged arrays - arrays of arrays
- Strings in C#: Same notation as in C, and similar escape characters
- Structs in C#: A value type like in C.

The most important differences are:

- Arrays: Rectangular arrays in C#
- Strings: No **`\0`** termination in C#
- Structs: Much expanded in C#. Structs can have methods.

A C programmer, who have experience with arrays, strings, and structs from C, will immediately feel comfortable with these types in C#. But such a C programmer will also quickly find out, that there are substantial new possibilities in C# that makes life easier.

Arrays and strings will be discussed in Section 6.4. Classes and structs are, basically, what the rest of the book is about. The story about classes starts in Chapter 10.

6.4. Arrays and Strings

Lecture 2 - slide 10

Arrays and strings are both non-simple types that are well-known by C programmers. In C# both arrays and strings are defined by classes. As we will see later, this implies that arrays and strings are represented as objects, and they are accessed via references. This stands as a contrast to C# structs, which are values and therefore not accessed via references.

The syntax of array declaration and initialization is similar in C and C#. In C, a string is a pointer to a the first character in the string, and it is declared of the type `char*`. C# supports a type named `string`. The notation of string constants is also similar in C and C#, although C# offers additional possibilities (the **`@"..."`** notation, see below).

The following summarizes important differences in between C and C# with respect to arrays and strings:

- Arrays in C# can be rectangular or jagged (arrays of arrays)
- In C#, an array is not a pointer to the first element
- Index out of bound checking is done in C#
- Strings are immutable in C#, but not in C
- In C# there are two kinds of string literals: **`"a string\n"`** and **`@"a string\n"`**

A multi-dimensional array in C is constructed as an array in which the elements are themselves arrays. Such arrays are known as jagged arrays in C#, because not all constituent arrays need to have the same size. In addition C# supports a new kind of arrays, namely rectangular arrays (of two or more dimensions). Such arrays are similar to arrays in Pascal.

C is inherently unsafe, in part because indexes out of bounds are not systematically caught at run-time. C# is safe in this respect. An index out of bound in a running C# program raises an exception, see Chapter 36.

C programmers may be puzzled by the fact that strings are immutable in C#. Once a string is constructed, it is not possible to modify the character elements of the string. This is also a characteristic of strings in Java. This makes it possible to share a given string in several contexts. The bookkeeping behind this is called *interning*. (You can, for instance, read about interning in the documentation of the static method `String.Intern`). In case mutable strings are necessary, the class `System.Text.StringBuilder` makes them available.

The well-known double quote string notation is used both in C and C#. Escape characters, prefixed with backslashes (such as in **`"\n"`**) are used in C as well and in C#. C# supports an alternative notation, called *verbatim string constants*, **`@"..."`** , in which the only escape notation is **`""`** which stands for the double quote character itself. Inside a verbatim string constant it possible to have newline characters, and all backslashes appear as backslash characters in the string. An example of a verbatim strings will be shown in Program 6.9.

Below, in Program 6.7 we will demonstrate a number of aspects of arrays in C#.

```
using System;

class ArrayStringDemo{

  public static void Main(){
     string[]  a1,
               a2 = {"a", "bb", "ccc"};

     a1 = new string[]{"ccc", "bb", "a"};

     int[,]    b1 = new int[2,4],
               b2 = {{1,2,3,4}, {5,6,7,8}};

     double[][] c1 = { new double[]{1.1, 2.2},
                       new double[]{3.3, 4.4, 5.5, 6.6} };

     Console.WriteLine("Array lengths. a1:{0} b2:{1} c1:{2}",
                        a1.Length, b2.Length, c1.Length);

     Array.Clear(a2,0,3);

     Array.Resize<string>(ref a2,10);
     Console.WriteLine("Lenght of a2: {0}", a2.Length);

     Console.WriteLine("Sorting a1:");
     Array.Sort(a1);
     foreach(string str in a1) Console.WriteLine(str);
  }

}
```

Program 6.7 *Demonstrations of array types in C#.*

We declare two variables, `a1` and `a2`, of the type `string[]`. In other words, `a1` and `a2` are both arrays of strings. `a1` is not initialized in its declaration. (Local variables in C# are not initialized to any default value). `a2` is initialized by means of an *array initializer*, namely `{"a", "bb", "ccc"}`. The length of the array is determined by the number of expressions within the pair of curly braces.

The arrays `b1` and `b2` are both rectangular 2 times 4 arrays.

The array `c1` is an example of a jagged array. `c1[1]` is an array of length 2. `c1[2]` is an array of length 4.

Next we try out the `Length` operation on `a1`, `b2` and `c1`. The result is a1:3 `b2:8 c1:2`. Please notice and understand the outcome.

Finally we demonstrate a number of additional operations on arrays: `Clear`, `Resize`, and `Sort`. These are all methods in the class `System.Array`.

The output of the array demo program is shown in Listing 6.8 (only on web).

Arrays, as discussed above, will be used in many of your future programs. But as an alternative, you should be aware of the collection classes, in particular the type parameterized, "generic" collection classes. These classes will be discussed in Chapter 45.

Now we will study a program example that illustrates uses of the type `string`.

```
1  using System;
2  
3  class ArrayStringDemo{
4  
5    public static void Main(){
6       string s1        = "OOP";
7       System.String s2 = "\u004f\u004f\u0050";   // equivalent
8       Console.WriteLine("s1 and s2: {0} {1}", s1, s2);
9  
10      string s3 = @"OOP on
11             the \n semester ""Dat1/Inf1/SW3""";
12      Console.WriteLine("\n{0}", s3);
13 
14      string s4 = "OOP on \n            the \\n semester \"Dat1/Inf1/SW3\"";
15      Console.WriteLine("\n{0}", s4);
16 
17      string s5 = "OOP E06".Substring(0,3);
18      Console.WriteLine("The substring is: {0}", s5);
19   }
20 
21 }
```

Program 6.9 *A demonstration of strings in C#.*

The strings `s1` and `s2` in Program 6.9 contain the same three characters, namely 'O', 'O', and 'P'.

Similarly, the strings referred by `s3` and `s4` are equal to each other (in the sense that they contain the same sequences of characters). As already mentioned above, the string constant in line 10-11 is a *verbatim string constant*, in which an escape sequence like `\n` denotes itself. In verbatim strings, only "" `has` a special interpretation, namely as a single quoute character.

Finally, in Program 6.9, the `Substring` operation from the class `System.String` is demonstrated.

The output of the string demonstration program in Program 6.9 is shown in Listing 6.10 (only on web).

Exercise 2.5. *Use of array types*

Based on the inspiration from the accompanying example, you are in this exercise supposed to experiment with some simple C# arrays.

First, consult the documentation of the class `System.Array`. Please notice the properties and methods that are available on arrays in C#.

Declare, initialize, and print an array of names (e.g. array of strings) of all members of your group.

Sort the array, and search for a given name using `System.Array.BinarySearch` method.

Reverse the array, and make sure that the reversing works.

Exercise 2.6. *Use of string types*

Based on the inspiration from the accompanying example, you are in this exercise supposed to experiment with some simple C# strings.

First, consult the documentation of the class `System.String` - either in your documentation browser or at `msdn.microsoft.com`. Read the introduction (remarks) to string which contains useful information! There exists a large variety of operations on strings. Please make sure that you are aware of these. Many of them will help you a lot in the future!

Make a string of your own first name, written with escaped Unicode characters (like we did for "OOP" in the accompanying example). If necessary, consult the unicode code charts (Basic Latin and Latin-1) to find the appropriate characters.

Take a look at the `System.String.Insert` method. Use this method to insert your last name in the first name string. Make your own observations about `Insert` relative to the fact that strings in C# are immutable.

6.5. Pointers and references

Lecture 2 - slide 11

Pointers are instrumental in almost all real-life C programs, both for handling dynamic memory allocation, and for dealing with arrays. Recall that an array in C is simply a pointer to the first element of the array.

References in C# (and Java) can be understood as a restricted form of pointers. C# references are never explicitly dereferenced, references are not coupled to arrays, and references cannot be operated on via the arithmetic C# operators; There are no pointer arithmetic in (the safe part of) C#. As a special notice to C++ programmers: References in C# have nothing to do with references in C++.

Here follows an itemized overview of pointers and references.

- Pointers
    - In normal C# programs: Pointers are not used
        - All the complexity of pointers, pointer arithmetic, dereferencing, and the address operator is not found in normal C# programs
    - In specially marked unsafe sections: Pointers can be used almost as in C.
        - Do not use them in your C# programs!
- References
    - Objects (instance of classes) are always accessed via references in C#
    - References are automatically dereferenced in C#
    - There are no particular operators in C# that are related to references

Program 6.11 shows some basic uses of references in C#. The variables `cRef` and `anotherCRef` are declared of type `C`. `C` happens to be an almost trivial class that we have defined in line 3-5. Classes are reference types in `C` (see Chapter 13). `cref` declared in in line 11 is assigned to `null` (a reference to nothing) in line 12. Next, in line 15, `cref` is assigned to a new instance of `C`. Via the reference in `cref` we can access the members `x` and `y` in the `C` object, see line 18. We can also pass a reference as a parameter to a function `F` as in line 19. This does not copy the referenced object when entering `F`.

```
1  using System;
2  
3  public class C {
4    public double x, y;
5  }
6  
7  public class ReferenceDemo {
8  
9    public static void Main(){
10 
11     C cRef, anotherCRef;
12     cRef = null;
13     Console.WriteLine("Is cRef null: {0}", cRef == null);
14 
15     cRef = new C();
16     Console.WriteLine("Is cRef null: {0}", cRef == null);
17 
18     Console.WriteLine("x and y are ({0},{1})", cRef.x, cRef.y);
19     anotherCRef = F(cRef);
20   }
21 
22   public static C F(C p){
23     Console.WriteLine("x and y are ({0},{1})", p.x, p.y);
24     return p;
25   }
26 
27 }
```

Program 6.11 *Demonstrations of references in C#.*

The output of Program 6.11 is shown in Listing 6.12 (only on web).

There is no particular complexity in normal C# programs due to use of references

6.6. Structs

Lecture 2 - slide 12

Structs are well-known by C programmers. It is noteworthy that arrays and structs are handled in very different ways in C. In C, arrays are deeply connected to pointers. Related to the discussion in this material, we will say that pointers are dealt with by *reference semantics*, see Section 13.1. Structs in C are dealt with by *value semantics*, see Section 14.1. Structs are copied by assignment, parameter passing, and returns. Arrays are not!

Let us now compare structs in C and C#:

- Similarities
  - Structs in C# can be used almost in the same way as structs in C
  - Structs are *value types* in both C and C#
- Differences
  - Structs in C# are almost as powerful as classes
    - Structs in C# can have operations (methods) in the same way as classes
    - Structs in C# cannot inherit from other structs or classes

In Program 6.13 we see a program with a struct called `Point`. The variable `p1` contains a point (3.0, 4.0). Because structs are value types, `p1` does not refer to a point. It *contains* the two coordinates of type `double` that represents the points. `p2` is uninitialized. In line 15, `p1` is copied into `p2`. This is a field-by-field (bit-by-bit) copy operation. No manipulation of references is involved. Finally we show the activation of a method `Mirror` on `p2`. Hereby the state of the second point is mutated to (-3,-4).

```
1  using System;
2  
3  public struct Point {
4    public double x, y;
5    public Point(double x, double y){this.x = x; this.y = y;}
6    public void Mirror(){x = -x; y = -y;}
7  } // end Point
8  
9  public class StructDemo{
10 
11  public static void Main(){
12    Point p1 = new Point(3.0, 4.0),
13          p2;
14 
15    p2 = p1;
16 
17    p2.Mirror();
18    Console.WriteLine("Point is: ({0},{1})", p2.x, p2.y);
19  }
20 }
```

Program 6.13 *Demonstrations of structs in C#.*

6.7. Operators

Lecture 2 - slide 13

Expressions in C# have much in common with expressions in C. Non-trivial expressions are built with use of operators. We summarize the operators in C# in Table 6.1. As it appears, there is a substantial overlap with the well-known operators i C. Below the table we will comment on the details.

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

Table 6.1 *The operator priority table of C#. Operators with high level numbers have high priorities. In a given expression, operators of high priority are evaluated before operators with lower priority. The associativity tells if operators at the same level are evaluated from left to right or from right to left.*

The operators shown in red are new and specific to C#. The operator **`new`** creates an instance (an object) of a class or it initializes a value of struct. We have already encountered `new` in some of the simple demo programs, for instance Program 6.11 and Program 6.13. See Section 12.2 for details on **`new`**. The operators **`is`**, **`as`** and **`typeof`** will not be discussed here. Please refer to Section 28.12 for details on these. The operations **`checked`** and **`uncheked`** relate to the safe and unsafe part of C# respectively. In this material we only deal with the safe part, and therefore these two C# operators can be disregarded. The **`default`** operator gives access to the default value of value types, see Section 12.3. The **`delegate`** operator is used for definition of anonymous functions, see Section 22.1. The unary **`true`** and **`false`** operators tell when a value in a user defined type is regarded as *true* or *false* respectively. See Section 21.2 for more details. The expression `x ?? y` is a convenient shortcut of `x != null ? x : y`. See Section 14.9 for more details on **??**. **=>** is the operator which is used to form lambda expressions in C#3.0, see Section 22.4.

A couple of C operators are not part of C#. The address operator `&` and the dereference operator `*` are not found in (the safe part of) C# (but they are actually available in the unsafe part of the language). They are both related to pointers, and as discussed in Section 6.5 pointers are not supported in (the safe part of) C#.

All the remaining operators should be familiar to the C programmer.

The operators listed above have fixed and predefined meanings when used together with primitive types in C#. On top of these it is possible to define new meanings of some of the operators on objects/values of your own types. This is called *operator overloading*, and it is discussed in more details in Chapter 21. The subset of overloadable operators is highlighted in Table 21.1.

6.8. Commands and Control Structures

Lecture 2 - slide 14

Almost all control structures in C can be used the same way in C#

Commands (also known as `statements`) are able to mutate the program state at run-time. As such, the most important command is the assignment. The commands constitute the "imperative heart" of the programming language. The control structures provide means for sequencing the commands.

The commands and control structures of C# form - to a large extent - a superset of the commands and control structures of C. Thus, the knowledge of commands and control structures in C can be used directly when learning C#.

As usual, we will summarize similarities and differences between C and C#. The similarities are the following:

- Expression statements, such as **`a = a + 5;`**
- Blocks, such as **`{a = 5; b = a;}`**
- **`if`**, **`if-else`**, **`switch`**, **`for`**, **`while`**, **`do-while`**, **`return`**, **`break`**, **`continue`**, and **`goto`** in C# are all similar to C

As in C, an expression becomes a command if it is followed by a semicolon. Therefore we have emphasized the semicolon above in the assignment `a = a + 5;`

As it will be clear from Program 6.15 below, the **switch** control structures in C and C# differ substantially.

The main differences between C and C# regarding control structures are the following:

- The C# **`foreach`** loop provides for easy traversal of all elements in a collection
- **`try-catch-finally`** and **`throw`** in C# are related to exception handling

- Some more specialized statements have been added: **`checked`**, **`unchecked`**, **`using`**, **`lock`** and **`yield`**.

The **foreach** control structures is an easy-to-use version of a for loop, intended for start-to-end traversal of collections. We will not here touch on **try-catch-finally** and **throw**. Please refer to our coverage of exception handling in Section 36.2 for a discussion of these.

Let us now look at some concrete program examples with control structures. In the examples below, program fragments shown in **red** color illustrate erroneous aspects. Program fragments shown in **green** are all right.

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

class IfDemo {

  public static void Main(){
    int i = 0;

    /*
    if (i){
      Console.WriteLine("i is regarded as true");
    }
    else {
      Console.WriteLine("i is regarded as false");
    }
    */

    if (i != 0){
      Console.WriteLine("i is not 0");
    }
    else {
      Console.WriteLine("i is 0");
    }
  }
}
```

Program 6.14 *Demonstrations of if.*

The **if-else** control structure has survived from C. Program 6.14 in reality illustrates a difference between handling of boolean values of C and C#. This has already been treated in Section 6.1. The point is that an expression of non-`bool` type (such the integer `i`) cannot be used as the controlling expression of an **if-else** control structure i C#.

Let us now look at a program with **switch** control structures. As already mentioned earlier there are a number of noteworthy differences between C and C# regarding **switch**.

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

class SwitchDemo {
  public static void Main(){
    int j = 1, k = 1;

    /*
    switch (j) {
      case 0:  Console.WriteLine("j is 0");
      case 1:  Console.WriteLine("j is 1");
12        case 2:  Console.WriteLine("j is 2");
13        default: Console.WriteLine("j is not 0, 1 or 2");
14      }
15      */
16
17      switch (k) {
18        case 0:  Console.WriteLine("m is 0"); break;
19        case 1:  Console.WriteLine("m is 1"); break;
20        case 2:  Console.WriteLine("m is 2"); break;
21        default: Console.WriteLine("m is not 0, 1 or 2"); break;
22      }
23
24      switch (k) {
25        case 0: case 1:  Console.WriteLine("n is 0 or 1"); break;
26        case 2: case 3:  Console.WriteLine("n is 2 or 3"); break;
27        case 4: case 5:  Console.WriteLine("n is 4 or 5"); break;
28        default: Console.WriteLine("n is not 1, 2, 3, 4, or 5"); break;
29      }
30
31      string str = "two";
32      switch (str) {
33        case "zero":  Console.WriteLine("str is 0"); break;
34        case "one":  Console.WriteLine("str is 1"); break;
35        case "two":  Console.WriteLine("str is 2"); break;
36        default: Console.WriteLine("str is not 0, 1 or 2"); break;
37      }
38    }
39  }
```

Program 6.15 *Demonstrations of switch.*

The first switch in Program 6.15 is legal in C, but it is illegal i C#. It illustrates the *fall through problem*. If `j` is 0, case 0, 1, 2, and `default` are all executed in a C program. Most likely, the programmer intended to write the second switch, starting in line 17, in which each `case` is broken with use of the `break` command. In C# the compiler checks that each branch of a switch never encounters the ending of the branch (and, thus, never falls through to the succeeding branch).

The third switch in the demo program shows that two or more cases can be programmed together. Thus, like in C, it is legal to fall trough empty cases.

The final switch shows that it is possible to switch on strings in C#. This is very convenient in many contexts! In C, the type of the switch expression must be integral (which means an integer, char, or an enumeration type).

Let us also mention that C# allows special goto constructs (**`goto case`** *constant* and **`goto default`**) inside a switch. With use of these it is possible to jump from one case to another, and it is even possible to program a loop inside a switch (by jumping from one case to an already executed case). It is recommended only to use these specialized goto constructs in exceptional situations, or for programming of particular patterns (in which it is natural to organize a solution around multiple branches that can pass the control to each other).

Next we will study a program that illustrates the **foreach** loop.

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

class ForeachDemo {
  public static void Main(){

    int[] ia = {1, 2, 3, 4, 5};
    int sum = 0;

    foreach(int i in ia)
      sum += i;

    Console.WriteLine(sum);
  }
}
```

Program 6.16 *Demonstrations of foreach.*

As mentioned above, **`foreach`** is a variant of a for loop that traverses all elements of a collection. (See how this is provided for in Section 31.6). In the example of Program 6.16 all elements of the array are traversed. Thus, the loop variable `i` will be 1, 2, 3, 4 and 5 in succession. Many efforts in C# have been directed towards supporting foreach on the collection types that you program yourself. Also notice that loop control variable, `i`, is declared inside the foreach construct. This cannot be done in a conventional for loop in C (although it can be done in C99 and in C++).

Finally, we will see an example of **try-catch**.

```
1  /* Right, Wrong */
2  using System;
3  
4  class TryCatchDemo {
5    public static void Main(){
6      int i = 5, r = 0, j = 0;
7  
8      /*
9      r = i / 0;
10     Console.WriteLine("r is {0}", r);
11     */
12 
13     try {
14       r = i / j;
15       Console.WriteLine("r is {0}", r);
16     } catch(DivideByZeroException e){
17         Console.WriteLine("r could not be computed");
18     }
19   }
20 }
```

Program 6.17 *Demonstrations of try catch.*

Division by 0 is a well-known cause of a run-time error. Some compilers are, in some situations, even smart enough to identify the error at compile-time. In Program 6.17 the erroneous program fragment never reaches the activation of `WriteLine` in line 10, because the division by zero halts the program.

The expression `i / j`, where `j` is 0, is embedded in a **try-catch** control structure. The division by zero raises an exception in the running program, which may be handled in the catch part. The `WriteLine` in line 17 is encountered in this part of the example. Thus, the program survives the division by zero. Later in the material, starting in Chapter 33, we discuss - in great details - errors and error handling and the use of **try-catch**.

Before we leave the assignments and control structure we want to mention the *definite assignment* rule in C#. The rule states that every declared variable must be assigned to a value before the variable is used. The compiler enforces the rule. Take a look at the program below.

```
1  using System;
2  
3  class DefiniteAssignmentDemo{
4  
5    public static void Main(){
6      int a, b;
7      bool c;
8  
9      if (ReadFromConsole("Some Number") < 10){
10        a = 1; b = 2;
11     } else {
12        a = 2;
13     }
14 
15     Console.WriteLine(a);
16     Console.WriteLine(b);  // Use of unassigned local variable 'b'
17 
18     while (a < b){
19       c = (a > b);
20       a = Math.Max(a, b);
21     }
22 
23 
24     Console.WriteLine(c);  // Use of unassigned local variable 'c'
25 
26   }
27 
28   public static int ReadFromConsole(string prompt){
29     Console.WriteLine(prompt);
30     return int.Parse(Console.ReadLine());
31   }
32 }
```

Program 6.18 *Demonstrations of definite assignment.*

The program declares three variables `a`, `b`, and `c` in line 6-7, without initializing them. Variable `b` is used in line 16, but it cannot be guarantied that the **`if-else`** control structure in line 9-13 assigns a value to the variable `b`. Therefore, using a *conservative approach*, the compiler complains about line 16. The error message is emphasized in the comment at the end of line 16.

Similarly, the variable `c` declared in line 7 is not necessarily assigned by the **`while`** control structure in line 18-21. Recall that if `a >= b` when we enter the while loop, the line 19 and 20 are never executed. This explains the error message associated to line 24.

The definite assignment rule, enforced by the compiler, implies that we never get run-time errors due to uninitialized variables. On the other hand, the rule also prevents some program from executing on selected input. If the number read in line 9 of Program 6.18 is less than 10 both `b` and `c` will be assigned when used in the `WriteLine` calls.

6.9. Functions

Lecture 2 - slide 15

Functions are the primary abstractions in C. In C# "function" (or "function member") is the common name of a variety of different kinds of abstractions. The most well-known of these is known as methods. The others are properties, events, indexers, operators, and constructors.

Functions in C# belong to types: classes or structs. Thus, functions in C# cannot be freestanding like in C. Functions are always found inside a type.

The conceptual similarities between functions in C and methods in C# are many and fundamental. In our context it is, however, most interesting to concentrate on the differences:

- Several different parameter passing techniques in C#
  - Call by value. For input. No modifier.
  - Call by reference. For *input and output* or *output only*
    - Input and output: Modifier **`ref`**
    - Output: Modifier **`out`**
    - Modifiers used both with formal and actual parameters
- Functions with a variable number of input parameters in C# (cleaner than in C)
- Overloaded function members in C#
- First class functions (delegates) in C#

In C all parameters are passed by value. However, passing a pointer by value in C is often proclaimed as *call by reference*. In C# there are several parameter passing modes: *call by value* and two variants of *call by reference* (`ref` and `out` parameters). The default parameter passing mode is call by value. *Call by reference* parameter passing in C (via pointers) is not the same as `ref` parameters in C#. `ref` parameters in C# are much more like Pascal `var` (variable) parameters.

In C it is possible, but messy, to deal with functions of a variable (or an arbitrary) number of arguments. In C# this is easier and cleaner. It is supported by the `params` keyword in a formal parameter list. An example is provided in Program 6.20.

A function in C is identified by its name. A method in C# is identified by its name together with the types of the formal parameters (the so-called *method signature*). This allows several methods with the same names to coexist, provided that their formal parameter types differ. A set of equally named methods (with different formal parameter types) is known as *overloaded* methods.

A function in C# can be handled without naming it at all. Such functions are known as delegates. Delegates come from the functional programming language paradigm, where functions most often are *first class objects*. Something of *first class* can be passed as parameters, returned as results from functions, and organized in data structures independent of naming. Delegates seem to be more and more important in the development of C#. In C# 3.0 the nearby concepts of *lambda expressions* and *expression trees* have emerged. We have much more to say about delegates later in this material, see Chapter 22.

```
/* Right, Wrong */

using System;


/*
public int Increment(int i){
  return i + 1;
}

public void Main (){
  int i = 5,
      j = Increment(i);
  Console.WriteLine("i and j: {0}, {1}", i, j);
} // end Main
```

```
*/

public class FunctionDemo {

  public static void Main (){
    SimpleFunction();
  }

  public static void SimpleFunction(){
    int i = 5,
        j = Increment(i);
    Console.WriteLine("i and j: {0}, {1}", i, j);
  }

  public static int Increment(int i){
    return i + 1;
  }
}
```

Program 6.19 *Demonstration of simple functions in C#.*

Program 6.19 shows elementary examples of functions (methods) in a C# program. The program text decorated with **red** color shows two functions, `Main` and `Increment`, outside of any type. This is illegal in C#.

Shown in **green** we again see the function `Increment`, now located in a legal context, namely inside the type (class) `FunctionDemo`. The function `SimpleFunction` calls `Increment` in a straightforward way. The function `Main` serves as *main program* in C#. It is here the program starts. We see that `Main` calls `SimpleFunction`.

```
using System;
public class FunctionDemo {

  public static void Main (){
    ParameterPassing();
  }

  public static void ValueFunction(double d){
    d++;}

  public static void RefFunction(ref double d){
    d++;}

  public static void OutFunction(out double d){
    d = 8.0;}

  public static void ParamsFunction(out double res,
                                    params double[] input){
    res = 0;
    foreach(double d in input) res += d;
  }

  public static void ParameterPassing(){
    double myVar1 = 5.0;
    ValueFunction(myVar1);
    Console.WriteLine("myVar1: {0:f}", myVar1);            // 5.00

    double myVar2 = 6.0;
    RefFunction(ref myVar2);
    Console.WriteLine("myVar2: {0:f}", myVar2);            // 7.00

    double myVar3;
    OutFunction(out myVar3);
    Console.WriteLine("myVar3: {0:f}", myVar3);             // 8.00

    double myVar4;
    ParamsFunction(out myVar4, 1.1, 2.2, 3.3, 4.4, 5.5);  // 16.50
    Console.WriteLine("Sum in myVar4: {0:f}", myVar4);
  }

}
```

Program 6.20   *Demonstration of parameter passing in C#.*

The four functions in Program 6.20, `ValueFunction`, `RefFunction`, `OutFunction`, and `ParamsFunction` demonstrate the different parameter passing techniques of C#.

The call-by-value parameter `d` in `ValueFunction` has the same status as a local variable in `ValueFunction`. Therefore, the call of `ValueFunction` with `myVar1` as actual parameter does not affect the value of `myVar1`. It does, however, affect the value of `d` in `ValueFunction`, but this has no lasting effect outside `ValueFunction`. In a nutshell, this is the idea of call by value parameters.

In `RefFunction`, the formal parameter `d`, is a **ref** parameter. The corresponding actual parameter must be a variable. And indeed it is a variable in our sample activation of `RefFunction`, namely the variable named `myVar2`. Inside `RefFunction`, the formal parameter `d` is an *alias* of the actual parameter (`myVar2`). Thus, the incrementing of `d` actually increments `myVar2`. Pascal programmers will be familiar with this mechanism (via var parameters) but C programmers have not encountered this before - at least not while programming in C.

`OutFunction` demonstrates the use of an **out** parameter. **out** parameters are similar to **ref** parameters, but only intended for data output from the function. More details of **ref** and **out** parameters appears in Section 20.6 and Section 20.7.

Notice that in C#, the keywords **ref** and **out** must be used both in the formal parameter list and in the actual parameter list. This is nice, because you will hereby spot the parameter passing mode in calls. In most other programming language it is necessary to consult the function definition to find out about the parameter passing modes of the parameters involved.

The first parameter of `ParamsFunction`, `res`, is an **out** parameter, intended for passing the sum of the `input` parameter back to the caller. The formal **param** parameter, `input`, must be an array. The similar actual parameters (occurring at the end of the actual parameter list) are inserted as elements into a new array, and bound to the formal parameter `input`. With this mechanism, an arbitrary number of "rest parameters" (of the same or comparable types) can be handled, and bundled into an array in the C# function, which is being called.

Program 6.21 (only on web) shows a class with four methods, all of which are named `F`. These functions are distinguished by different formal parameters, and by different parameter passing modes. Passing an integer value parameter activates the first `F`. Passing a double value parameter activates the second `F`. Passing a double and a bool (both as values) activates the third `F`. Finally, passing an integer **ref** parameter activates the fourth `F`.

6.10. Input and output

Lecture 2 - slide 16

In C, the functions `printf` and `scanf` are important for handling output to the screen, input from the keyboard, and file IO as well. It is therefore interesting for C programmers to find out how the similar facilities work in C#.

In C#, the `Console` class encapsulates the streams known as *standard input* and *standard output*, which per default are connected to the keyboard and the screen. The various write functions in the `Console` class are quite similar to the `printf` function in C. The `Console` class' read functions are not as advanced as `scanf` in C. There is not direct counterpart of the C `scanf` function in C#.

First, in Program 6.22 we will study uses of the `Write` and `WriteLine` functions.

```
/* Right, Wrong */

using System;
public class OutputDemo {

// Placeholder syntax: {<argument#>[,<width>][:<format>[<precision>]]}

  public static void Main(){
    Console.Write(    "Argument number only: {0} {1}\n", 1, 1.1);
//  Console.WriteLine("Formatting code d: {0:d},{1:d}", 2, 2.2);

    Console.WriteLine("Formatting codes d and f: {0:d}  {1:f}", 3, 3.3);
    Console.WriteLine("Field width: {0,10:d}  {1,10:f}", 4, 4.4);
    Console.WriteLine("Left aligned: {0,-10:d}  {1,-10:f}", 5, 5.5);
    Console.WriteLine("Precision: {0,10:d5}  {1,10:f5}", 6, 6.6);
    Console.WriteLine("Exponential: {0,10:e5}  {1,10:e5}", 7, 7.7);
    Console.WriteLine("Currency: {0,10:c2}  {1,10:c2}", 8, 8.887);
    Console.WriteLine("General: {0:g}  {1:g}", 9, 9.9);
    Console.WriteLine("Hexadecimal: {0:x5}", 12);

    Console.WriteLine("DateTime formatting with F: {0:F}", DateTime.Now);
    Console.WriteLine("DateTime formatting with G: {0:G}", DateTime.Now);
    Console.WriteLine("DateTime formatting with T: {0:T}", DateTime.Now);
  }
}
```

Program 6.22 *Demonstrations of Console output in C#.*

Like `printf` in C, the methods `Write` and `WriteLine` accept a control string and a number of additional parameters which are formatted and inserted into the control string. `Write` and `WriteLine` actually rely on an underlying `Format` method in class `String`. Notice that a there exists many overloaded `Write` and `WriteLine` methods in the class `Console`. Here we concentrate of those that take a string - the control string - as the first parameter.

The following call of `printf` in C

```
printf("x: %d, y: %5.2f, z: %Le\n", x, y, z);
```

is roughly equivalent to the following call of of `WriteLine` in C#

```
Console.WriteLine("x: {0:d}, y: {1,5:F2}, z: {2:E}", x, y, z);
```

The equivalence assumes that `x` is of type int, `y` is a float, and that `z` is a long double.

The general syntax of a *placeholder* (the stuff in between pairs of curly braces) in a C# formatting string is

{<argument#>[,<width>][:<format>[<precision>]]}

where `[...]` denotes optionality (zero or one occurrence).

C programmers do often experience strange and erroneous formatting of output if the conversion characters (such as `d`, `f`, and `e` in the example above) are inconsistent with the actual type of the corresponding variables or expressions (`x`, `y`, and `z` in the example). In C#, such problems are caught by the compiler, and as such they do not lead to wrong results. This is a much needed improvement.

Let us briefly explain the examples in line 9-23 of Program 6.22. In line 9 the default formatting is used. This corresponds to the letter code `g`. In line 10 an error occurs because the code `d` only accepts integers. The number 2.2 is not an integer. In line 13 we illustrate use of width 10 for an integer and a floating-point number. Line 14 is similar, but it uses left justification (because the width is negative). Line 15 illustrates use of the precision 5 for an integer and a floating-point number. In line 16 we format two numbers in exponential (scientific) notation. In line 17 we illustrate formatting of currencies (kroner or dollars, for instance, dependent on the culture setting). Line 18 corresponds to line 9. Line 19 calls for hexadecimal formatting of a number.

One way to learn more about output formatting is to consult the documentation of the static method `Format` in class `System.String`. From there, goto Formatting Overview. Later in this material, in Section 31.7, we will see how we can program custom formatting of our own types.

The last three example lines in Program 6.22 illustrate formatting of objects of type `DateTime` in C#. Such objects represent at point in time. In the example, the expression `DateTime.Now` denotes the current point in time.

The output of Program 6.22 is shown in Listing 6.23 (only on web).

We now switch from output to input.

```
/* Right, Wrong */

using System;
public class InputDemo {

  public static void Main(){
    Console.Write("Input a single character: ");
    char ch = (char)Console.Read();
    Console.WriteLine("Character read: {0}", ch);
    Console.ReadLine();

    Console.Write("Input an integer: ");
    int i  = int.Parse(Console.ReadLine());
    Console.WriteLine("Integer read: {0}", i);

    Console.Write("Input a double: ");
    double d  = double.Parse(Console.ReadLine());
    Console.WriteLine("Double read: {0:f}", d);
  }
}
```

Program 6.24 *Demonstrations of Console input in C#.*

In Program 6.24 `Console.Read()` reads a single character. The result returned is a positive integer, or -1 if no character can be read (typically because we are located at the end of an input file). `Read` blocks until **enter** is typed. Non-blocking input is also available via the method `Console.ReadKey`. The expression `Console.ReadLine()` reads a line of text into a string. The last two, highlighted examples show how to read a text string and, via the `Parse` method in type `int` and `double`, to convert the strings read to values of type `int` and `double` respectively. Notice that `scanf` in C can take hand of such cases.

The output of Program 6.24 is shown in Listing 6.25 (only on web).

Later in this material we have much more to say about input and output in C#. See Chapter 37 - Chapter 39 . The most important concept, which we will deal with in these chapters, is the various kinds of streams in C#.

6.11. Comments

Lecture 2 - slide 17

We finalize our comparison of C and C# with an overview of the different kinds of C# comments. Recall that C only supports delimited comments (although C programmers also often use single-line comments, which actually is used in C++ and in newer versions of C (C99)).

C# supports two different kinds of comments and XML variants of these:

- **Single-line comments like in C++**
  `// This is a single-line comment`
- **Delimited comments like in C**
  `/* This is a delimited comment */`
- **XML single-line comments:**
  `/// <summary> This is a single-line XML comment </summary>`
- **XML delimited comments:**
  `/** <summary> This is a delimited XML comment </summary> */`

XML comments can only be given before declarations, not inside other fragments. XML comments are used for documentation of types. We have much more to say about XML comments in our discussion of documentation of C# programs. Delimited C# comments cannot be nested.

6.12. References

[Decimal-floating-point]	Decimal Floating Point in .NET http://www.yoda.arachsys.com/csharp/decimal.html

7. C# in relation to Java

C# is heavily inspired by Java. In this section we will, at an overall level, compare Java and C#. The goal of this relatively short chapter is to inform Java programmers about similarities and differences in relation to C#. It is recommended that you already have familiarized yourself with C# in relation to C, as covered in Chapter 6.

In this chapter 'Java' refers to version to 5.0 and 'C#' refers to C# version 2.0.

7.1. Types

Lecture 2 - slide 20

In Java, types can be defined by classes and interfaces. This is also possible in C#. In addition, C# makes it possible to define structs and delegates to which there are no counterparts in Java. Java supports sophisticated, class based enumeration types. Enumeration types in C# are simpler, and relative close to enumeration types in C. - This is the short story. After the itemized summary, we will compare the two languages more carefully.

The similarities and differences with respect to types can be summarized in this way:

- Similarities.
    - Classes in both C# and Java
    - Interfaces in both C# and Java
- Differences.
    - Structs in C#, not in Java
    - Delegates in C#, not in Java
    - Nullable types in C#, not in Java
    - Class-like Enumeration types in Java; Simpler approach in C#

If you have written a Java program with classes and interfaces, it is in most cases relatively easy to translate the program to C#. In this material we discuss classes in C# in Chapter 11 an interfaces in Chapter 31

There are no structs in Java. Structs in C# are, at the outset, similar to structs in C. (See Section 6.6 for a discussion of C structs versus C# structs). Your knowledge of C structs is a good starting point for working with structs in C#. However, structs in C# are heavily extended compared with C. As an important observation, C# structs have a lot in common with classes. Most important, there are operations (methods) and constructors in both C# classes and C# structs. It is also possible to control the visibility of data and operations in both classes and structs. Structs in C# are value types, in the meaning that instances of structs are contained in variables, and copied by assignments and in parameter passings. In contrast, classes are reference types, in the meaning that instances of classes are accessed by references. Instances of classes are not copied in assignments and in parameter passings. For more details on structs see Chapter 14, in particular Section 14.3.

Delegates in C# represents a type of methods. A delegate object can contain a method. More correctly, a delegate can contain a reference to a method. It can actually contain several such references. With use of delegates it becomes possible to treat methods as *data*, in the same way as instance of classes represent data. We can store a method in a variable of delegate type. We can also pass a method as a parameter to another method. In Java it is not possible pass a method `M` as a parameter to another method. If we need to pass `M`, we have to pass an object of class `C` in which `M` is a method. Needless to say, this is a complicated and contrived way of using function parameters. - In C#, delegates are the foundation of events (see Chapter 23), which, in turn, are instrumental to programming of graphical user interfaces in C# and certain design patters, not least the ***Observer*** (see Section 24.1). For more details on delegate types see Chapter 22.

Nullable types relate to value types, such as structs. A variable of a struct type `S` cannot contain the value `null`. In contrast, a variable of class type `C` can contain the value `null`. The nullable `S` type, denoted `S?`, is a variant of `S` which includes the `null` value. For more details, see Section 14.9.

Enumeration types in both C# and Java allow us to associate symbolic constants to values in an integer type. We demonstrated enumeration types in C# in Section 6.2 of the previous chapter. In Java, an enumeration type is a special form of a class. Each enumerated value is an instance of this special class. Consequently, an enumeration type in Java is a reference type. In C# an enumeration type is a value type. - As a Java-historic remark, enumeration types did not exist in early versions Java. Enumeration types were simulated by a set of `final static` variables (one `final static` variable for each value in the enumeration type). The support of enumeration types shifted dramatically in Java 1.5: from almost no support in previous versions to heavy support via special classes. It is remarkable that the Java designers have chosen to use so many efforts on enumeration types!

7.2. Operations

Lecture 2 - slide 21

Operations in Java programs are defined by methods that belong to classes. This is our only possibility of defining operations in Java. In C# there exists several additional possibilities. In this material we have devoted an entire lecture 'Data Access and Operations' (from Chapter 17 to Chapter 24) to these issues.

The similarities and differences with respect to operations can be summarized in this way:

- Similarities: Operations in both Java and C#
    - Methods
- Differences: Operations only in C#
    - Properties
    - Indexers
    - Overloaded operators
    - Anonymous methods

As already mentioned above, C# methods can be defined in both classes and structs. It is not possible to define local methods in methods - neither in C# or Java. The closest possibility in C# is use of anonymous methods, see below.

Properties provide for getters and setters of fields (instance variables as well as class variables - static variables) in C#. In Java it is necessary to define methods for getting and setting of instance variables. A single property in C# can either be a getter, a setter, or both. From an application point of view, properties are used as though we access of the variables of a class/object directly (as it would be possible if the variables were public). For more details on properties see Chapter 18.

Indexers can be understood as a special kind of properties that define array-like access to objects and values in C#. The notation `a[i]` and `a[i] = x` is well-know when the name `a` denotes an array and if `i` is an integer. In C# we generalize this notation to arbitrary instances of classes and structs denoted by `a`. With an indexer we program the meaning of accessing the i'th element of `a` (`a[i]`) and the meaning of setting the i'th element of `a` (`a[i] = ...`). Indexers are discussed in Chapter 19.

In most languages, the operators like `+`, `-`, `<`, and `&` have fixed meanings, and they only work with the simple types (such as `int`, `bool`, `char`, etc). We reviewed the C# operators in Section 6.7, and as it appears, operators in C, Java, and C# have much in common. In Java, the operators only work for certain predefined types, and you cannot change the meaning of these operators. In C# it is possible to use the existing operator symbols for operations in your own types. (You cannot invent new operator symbols, and you cannot change the precedence nor the associativity of the symbols). We say that the operators in C# can be *overloaded*. For instance, in C# it would be possible to define the meaning of `aBankAccount + aTriangle`, where `aBankAccout` refers to an instance of class `BankAccount` and `aTriangle` refers to an instance of class `GeometricShape`. When the existing operator symbols are natural in relation to our own classes, the idea of overloaded operators is great. In other situations, overloaded operators do not add much value.

We are used to a situation where procedures, functions, and methods have names. In both Java and C# we can define named methods in classes. In C#, we can also define named methods in structs. In C# it is possible to define non-named methods as well. As part of arbitrary expressions, we can create a function or method. Such a function or method is called a delegate. As indicated by the name, delegates are closely related to delegate types, as discussed above in Section 7.1. For more details on this topic see Chapter 22 and in particular the program examples of Section 22.1.

7.3. Other substantial differences

Lecture 2 - slide 22

In addition to the overall differences in the area of types and operations, as discussed in the two previous sections, there are a number of other substantial differences between Java and C#. The most important of these differences are summarized below.

- Program organization
  - No requirements to source file organization in C#
- Exceptions
  - No *catch or specify* requirement in C#
- Nested and local classes
  - Classes inside classes are static in C#: No *inner classes* like in Java
- Arrays
  - Rectangular arrays in C#
- Virtual methods
  - Virtual as well as non-virtual methods in C#

In Java there is a close connection between classes and sources files. It is usually recommended that there is only one class per file, but the rule is actually that there can be one public and several non-public classes per Java source file. The proper name of the source file should be identical to the name of the public class. Likewise, there is a close connection between packages and directories. A package consists of the classes whose source files belong to a given directory. - The organization of C# programs is different. In C# there is no connection between the names of classes and the name of a C# source files. A source file can contain several classes. Instead of packages, C# organizes types in namespaces and assemblies. Namespaces are tangible, as they are represented syntactically in the source files. A namespace contains types and/or recursively other (nested) namespaces. Assemblies are produced by the C# compiler. Assemblies represent a 'packaging' mechanism, and assemblies are almost orthogonal to both the file/directory organization and the namespace organization. As it appears, C# uses a much more complex - and presumably more powerful - program organization than Java. We discuss organization of C# programs in Chapter 15 at the end of the lecture about Classes.

Java supports both *checked exceptions* and *unchecked exceptions*, but it is clearly the ideal of Java to work with checked exceptions. (Unchecked exception is also known as RuntimeExceptions). It is natural to ask about the difference. A checked exception must either be handled in the method `M` in which is occurs, or the method `M` must declare that an activation of `M` can cause an exception (of a given type) which callers of `M` need to care about. This is sometimes called the *catch or specify principle*. The most visible consequence of this principle is that Java methods, which do not handle exceptions, must declare that they **throws** specific types of exceptions. Thus, the signature of Java methods include information about the kind errors they may cause. - C# does not adhere to the *catch or specify principle*. In C# all exceptions correspond to the so-called RuntimeExceptions in Java. Exceptions in C# are discussed in Chapter 36.

Java is stronger than C# with respect to class nesting. Both Java and C# support *static nested classes* (using Java terminology). In this setup, the innermost class can only refer to static members of the outer class. In contrast to C#, Java also supports inner *classes*. An instance of an inner class has a reference the instance of the outer class. Inner classes have implications to the object structure. Static nested classes have no such implications. In addition, Java supports *local classes* that are local to a method, and *anonymous classes* which are instantiated on the fly. C# does not.

In both Java and C# it is possible to work with arrays in which the elements themselves are arrays, and so on recursively. Using C# terminology, this is called *jagged arrays*, because it facilitates multi-dimensional arrays of irregular shapes. In contrast to Java, C# in addition supports *rectangular arrays*, in which all rows are of equal lengths. We have already discussed jagged and rectangular arrays in our comparison with C in Section 6.4.

Virtual methods relate to redefinition of methods in class hierarchies (inheritance). In Java all methods are virtual. What this means (for C#) is explained in details in Section 28.14. In C# it is possible to have both virtual and non-virtual methods. This complicates the understanding of inheritance quite a bit. There are, however, good reasons to support both. In Section 32.9 we will review a prominent example where the uniform use of virtual methods in Java runs into trouble.

8. C# in relation to Visual Basic

This chapter is intended for students who have a background in imperative Visual Basic programming. The goal of this chapter is to make the transfer from Visual Basic to C# as easy as possible. We do that by showing and discussing a number of equivalent Visual Basic and C# programs. In this chapter Visual Basic programs are shown on a blue background, and C# programs are shown on a green background. The discussion of equivalent Visual Basic and C# program is textually organized in between the two programs.

In this chapter 'Visual Basic' refers to the version of Visual Basic supported by the .Net Framework version 2.0.

In this edition the comparison of Visual Basic and C# is only available in the web version of the material.

9. C# Tools and IDEs

Many potential C# programmers will be curious about tools and environments (IDEs) for C# programming. Therefore we will, briefly, enumerate the most obvious possibilities. We will mention possibilities in both Windows and Unix.

9.1. C# Tools on Windows

Lecture 2 - slide 41

Windows is the primary platform of C#. This is due to the fact that C# is a Microsoft language.

Microsoft supplies several different set of tools that support the C# programmer:

- .NET Framework SDK 3.5
    - "Software Development Kit"
    - Command line tools, such as the C# compiler **`csc`**
- Visual C# Express
    - IDE - An Integrated Development Environment
    - A C# specialized, *free* version of Microsoft Visual Studio 2008
- Visual Studio
    - IDE - An Integrated Development Environment
    - The professional, *commercial* development environment for C# and other programming languages

The .Net Standard Development Kit (SDK) supports the raw tools, including a traditional C# compiler. Although many programmers today use contemporary IDEs such as Visual Studio or Eclipse, I find it important that all programmers understand the basic and underlying activation of the compiler.

The Visual C# Express edition is a free (gratis) variant of Visual Studio, explicitly targeted at students and other newcomers to C#. There are video resources [cs-video-resources] available for getting started with C# 2008 Express. The experience you get with Visual C# Express can immediately be transferred to Visual studio. The two IDEs are very similar.

Visual Studio is the commercial flagship environment of C# programming. You will have to buy Visual Studio if you want to use it. Notice, however, that many universities have an academic alliance with Microsoft that provides access to Visual Studio and other Microsoft software.

9.2. C# Tools on Unix

Lecture 2 - slide 42

The MONO project provides tools for C# development on Linux, Solaris, Mac OS X, Unix in general, and interesting enough also on Windows. MONO is the choice if you swear to the Linux platform.

Let us summarize the MONO resources, available to the Linux people:

- MONO
    - An *open source* project (sponsored by Novell)
    - Corresponds the the Microsoft SDK
    - Based on ECMA specifications of C# and the Common Language Infrastructure (CLI) Virtual Machine
    - Command line tools
    - Compilers: **`mcs`** (C# 1.5) and **`gmcs`** (C# 2.0)
- MONO on cs.aau.dk
    - Mono is already installed on the application servers at cs.aau.dk
- MONO on your own Linux machine
    - You can install MONO yourself if you wish
- MonoDevelop
    - A GNOME IDE for C#

For good reasons, the MONO CLI is not as updated as the .NET solutions. MONO will most probably always be at least one step behind Microsoft.

9.3. References

[Cs-video-resources] C# Express Video Lectures
http://msdn.microsoft.com/en-us/beginner/bb964631.aspx

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*.

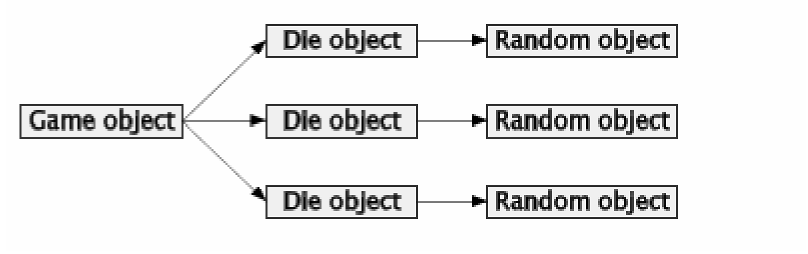

Figure 10.1 *Interacting Game, Die, and Random objects. The Game object is a client of the Die objects, which in turn are clients of the Random objects.*

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.

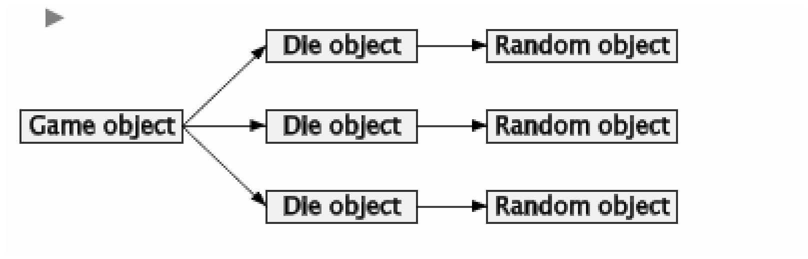

Figure 10.2 *Interacting* **`Game`**, **`Die`**, *and* **`Random`** *objects*

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.

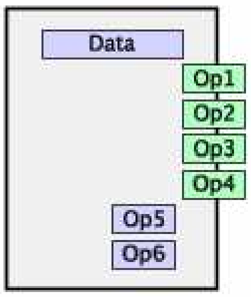

Figure 11.1 *A class and its interface to other classes. The interface is often called the client interface. In this illustration the operations Op1, Op2, Op3, and Op4 form the client interface of the class.*

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.

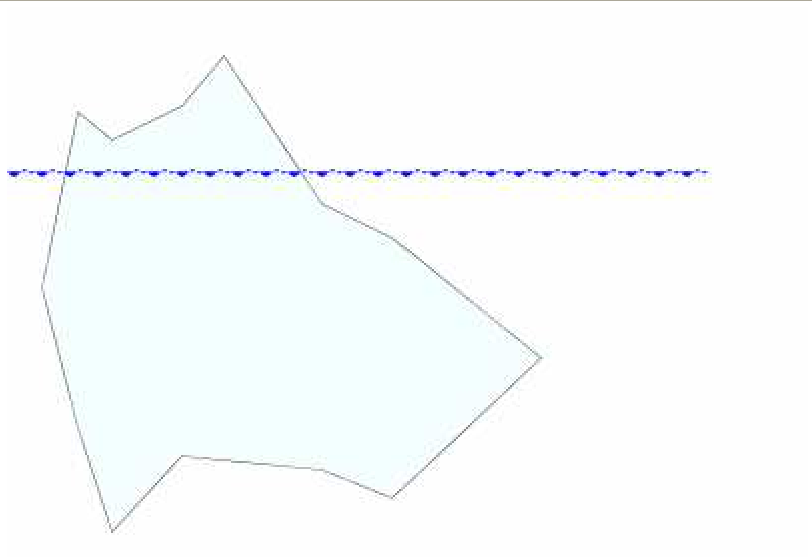

Figure 11.2 *An Iceberg. Only a minor fraction of the iceberg is visible above water. In the same way, only a small part of the details of a class should be visible from other classes.*

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).

```
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 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.

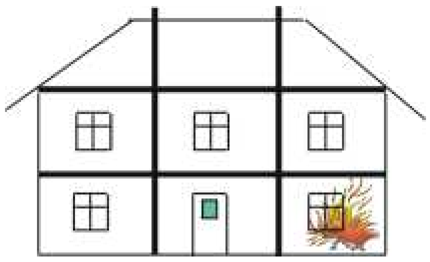

Figure 11.3 *A house - with firewalls - on fire. The fire is not likely to spread to other apartments because of the solid firewalls.*

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`.

```
// A very simple point with public data representation.
// NOT RECOMMENDED because of public data representation.

using System;

public class Point {
  public double x, y;

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

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

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

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.

```
using System;

public class BankAccount {

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

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


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

   public BankAccount(string owner, double interestRate,
                      decimal balance) {
```

```
    this.interestRate = interestRate;
    this.owner = owner;
    this.balance = balance;
  }


  // Remaining methods are not shown here
}
```

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.

```
using System;

public class BankAccountClient {

  public static void Main(){
    BankAccount a1 = new BankAccount("Kurt", 0.02),
                a2 = new BankAccount("Bent", 0.03),
                a3 = new BankAccount("Thomas", 0.02);

    a1.Deposit(100.0M);
    a2.Deposit(1000.0M); a2.AddInterests();
    a3.Deposit(3000.0M); a3.AddInterests();

    Console.WriteLine(a1);   // 100 kr.
    Console.WriteLine(a2);   // 1030 kr.
    Console.WriteLine(a3);   // 3060 kr.
  }

}
```

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).

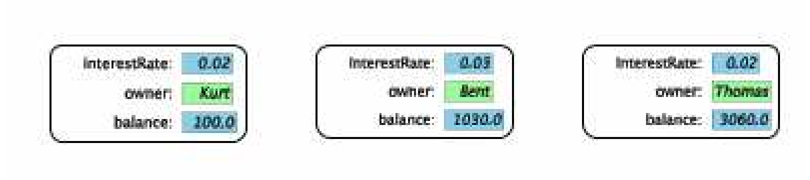

Figure 11.4 *Three objects of class BankAccount, each holding three instance variables `interestRate`, `owner`, and `balance`. The values of variables are determined by the bank account transactions that we programmed in the class `BankAccountClient`. The state of the variables is shown relative to the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

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

  public static long NumberOfAccounts (){
    return nextAccountNumber;
  }

  public static BankAccount GetAccount (long accountNumber){
     foreach(BankAccount ba in accounts)
       if (ba.accountNumber == accountNumber)
          return ba;
     return null;
  }

  // Some BankAccount methods are not shown in this version

}
```

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

```
using System;

public class BankAccountClient {

  public static void Main(){
    BankAccount a1 = new BankAccount("Kurt", 0.02),
                a2 = new BankAccount("Bent", 0.03),
                a3 = new BankAccount("Thomas", 0.02);

    a1.Deposit(100.0M);
    a2.Deposit(1000.0M); a2.AddInterests();
    a3.Deposit(3000.0M); a3.AddInterests();

    BankAccount a = BankAccount.GetAccount(2);
    if (a != null)
      Console.WriteLine(a);
    else
      Console.WriteLine("Cannot find account 2");
  }

}
```

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.

```
using System;

public class BankAccountClient {

  BankAccount
    a1 = new BankAccount("Kurt", 0.02),     // Error:
    a2 = new BankAccount("Bent", 0.03),     // An object reference is
    a3 = new BankAccount("Thomas", 0.02);   // required for the
```

```
                                                    // nonstatic field
  public static void Main(){

    a1.deposit(100.0);
    a2.deposit(1000.0); a2.addInterests();
    a3.deposit(3000.0); a3.addInterests();

    Console.WriteLine(a1);
    Console.WriteLine(a2);
    Console.WriteLine(a3);
  }

}
```

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,
                  cb = ca + 1;

  private readonly double roa = 7.0,
                          rob = Math.Log(Math.E);

  private readonly BankAccount
                  roba = new BankAccount("Anders");

  public ConstDemo(){   // CONSTRUCTOR
    roa = 8.0;
    roba = new BankAccount("Tim");
  }

  public static void Main(){
    ConstDemo self = new ConstDemo();
    self.Go();
  }

  public void Go(){
    roba.Deposit(100.0M);
  }
}
```

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.

```
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.

```
using System;

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

  public Die (Die d){
    numberOfEyes = d.numberOfEyes;
    randomNumberSupplier = d.randomNumberSupplier;
    maxNumberOfEyes = d.maxNumberOfEyes;
  }

  public Die (): this(6){}

  public Die (int maxNumberOfEyes){
    randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
    this.maxNumberOfEyes = maxNumberOfEyes;
    numberOfEyes = randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
  }

  // Die methods here

}
```

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.

13. Reference Types

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

13.1. Reference Types

Lecture 4 - slide 2

A class is a *reference type*

Objects instantiated from classes are accessed by references

The objects are allocated on *the heap*

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

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

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

13.2. Illustration of variables of reference types

Lecture 4 - slide 3

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

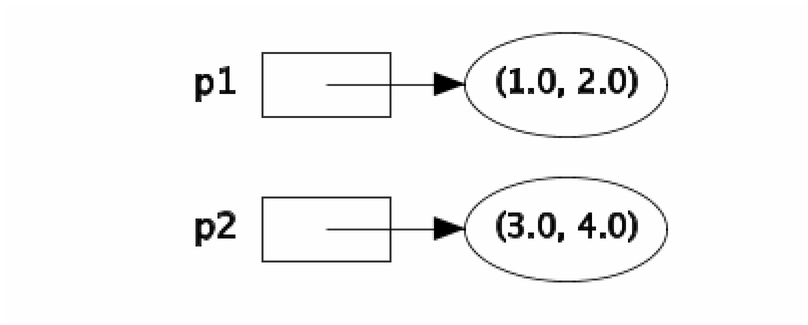

Figure 13.1   *Variables of reference types. The situation before the assignment p1 = p2.*

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

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

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

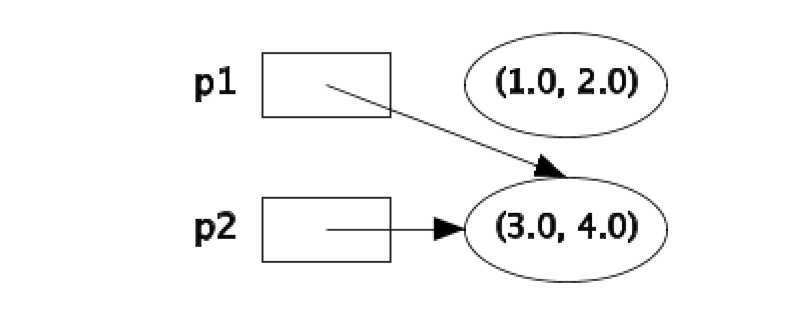

Figure 13.2   *Variables of reference types. The situation after the assignment p1 = p2.*

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

13.3. Overview of reference types in C#

Lecture 4 - slide 4

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

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

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

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

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

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

13.4. Comparing and copying objects via references

Lecture 4 - slide 5

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

Do we compare references or the referenced objects?

Do we copy the reference or the referenced object?

How deep do we copy objects that reference other objects?

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

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

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

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

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

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

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

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

An assignment of the form **`var = obj1`** copies a reference

A comparison of the form **`obj1 == obj2`** compares references (unless overloaded)

13.5. Equality in C#

Lecture 4 - slide 6

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System;

class WorderingAboutEquality{

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

    object m = 5,
           n = 5;

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

}
```

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

14. Value Types

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

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

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

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

14.1. Value types

Lecture 4 - slide 8

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

A variable of value type contains its value

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

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

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

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

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

14.2. Illustration of variables of value types

Lecture 4 - slide 9

Assume that the type Point is a value type

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

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

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

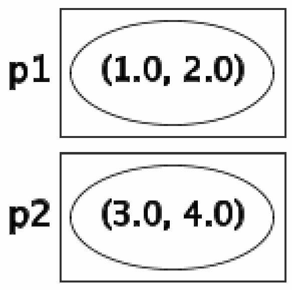

Figure 14.1 *Variables of value types. The situation before the assignment p1 = p2.*

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

p1 = p2;
```

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

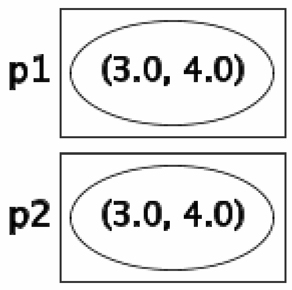

Figure 14.2 *Variables of value types. The situation after the assignment p1 = p2.*

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

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

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

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

14.3. Structs in C#

Lecture 4 - slide 10

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

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

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

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

```
using System;

public struct Point {
  private double x, y;

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

  public double Getx (){
    return x;
  }

  public double Gety (){
    return y;
  }

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

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

Program 14.2 *Struct Point.*

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

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

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

```
using System;

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

public struct Card {
  private CardSuite suite;
  private CardValue value;

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

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

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

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

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

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

Program 14.3 *Struct Card.*

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

```
using System;

public class PlayingCardClient{

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

    c4 = c1;  // Copies c1 into c4

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

}
```

Program 14.4   *A client of struct Card.*

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

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

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

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

14.4. Structs and Initialization

Lecture 4 - slide 11

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

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

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

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

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

Program 14.5 *Fields in structs cannot have initializers.*

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

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

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

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

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

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

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

14.5. Structs versus classes

Lecture 4 - slide 12

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

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

14.6. Examples of mutable structs in C#

Lecture 4 - slide 13

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

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

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

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

```
using System;

public class Application{

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

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

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

}
```

Program 14.7   *Moving a point by mutation.*

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

```
using System;

public struct Point {
  private double x, y;

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

  public double Getx (){
    return x;
  }

  public double Gety (){
    return y;
  }

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

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

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

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

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

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

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

Listing 14.11 *Output from the application.*

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

14.7. Examples of immutable structs in C#

Lecture 4 - slide 14

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

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

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

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

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

Program 14.12 *The struct Point - immutable.*

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

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

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

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

```
using System;

public class Application{

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

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

}
```

Program 14.13 *Application the struct Point - immutable.*

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

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

Listing 14.14 *Output from the application.*

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

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

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

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

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

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

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

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

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

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

14.8. Boxing and Unboxing

Lecture 4 - slide 15

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

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

Unboxing involves an unwrapping of a value from an object

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

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

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

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

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

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

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

The output of the program is shown in Listing 14.16.

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

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

14.9. Nullable types

Lecture 4 - slide 16

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

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

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

Nullable types provide a solution to the following desire:

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

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

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

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

```
public class IntSequence {

  private int[] sequence;

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

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

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

  // Other useful sequence methods

}
```

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

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

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

Program 14.19 *A client of IntSequence.*

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

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

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

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

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

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

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

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

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

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

15. Organization of C# Programs

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

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

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

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

15.1. Program Organization

Lecture 4 - slide 18

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

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

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

15.2. Examples of Program Organization

Lecture 4 - slide 19

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

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

using System;

public class C {

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

}
```

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

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

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

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

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

namespace N1 {
  public class C1{};
}


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

using System;

namespace Intro{

  internal class A {

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

  public class B {

    private A var = new A();

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

}
```

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

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

using System;

namespace Intro{

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

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

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

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

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

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

15.3. Namespaces and Visibility

Lecture 4 - slide 20

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

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

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

15.4. Namespaces and Assemblies

Lecture 4 - slide 21

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

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

15.5. References

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

16. Patterns and Techniques

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

16.1. Design Patterns

Lecture 4 - slide 23

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

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

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

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

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

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

16.2. Object-oriented Design Patterns

Lecture 4 - slide 24

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

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

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

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

16.3. The Singleton pattern

Lecture 4 - slide 25

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

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

Solution: The singleton design pattern

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

```
public class Singleton{

  // Instance variables

  private static Singleton uniqueInstance = null;

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

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

    return uniqueInstance;
  }

  // Methods

}
```

Program 16.1 *A template of a singleton class.*

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

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

```
using System;

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

  private static Die uniqueInstance = null;

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

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

    return uniqueInstance;
  }

  // Die methods: Toss and others

}
```

Program 16.2 *A singleton Die class.*

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

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

```
using System;

class diceApp {

  public static void Main(){

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

```

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

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

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

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

}
```

Program 16.3 *Application of the singleton Die class.*

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

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

16.4. Factory methods

Lecture 4 - slide 27

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

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

```
using System;

public class Point {

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

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

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

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

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

  // Remaining Point operations not shown
}
```

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

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

```
using System;

public class Point {

  public enum PointRepresentation {Polar, Rectangular}

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

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

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

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

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

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

  // Remaining Point operations not shown
}
```

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

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

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

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

16.5. Privacy Leaks

Lecture 4 - slide 29

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

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

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

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

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

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

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

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

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

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

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

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

Program 16.9 *A Mutable Date class.*

```
public class Person{

  private string name;
  private Date dateOfBirth, dateOfDeath;

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

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

  public Date DateOfBirth{
    get {return dateOfBirth;}
  }

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

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

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

}
```

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

```
using System;

class Client{

  public static void Main(){

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

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

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

}
```

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

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

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

Exercise 4.3. *Privacy Leaks*

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

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

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

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

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

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

16.6. References

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

17. Accessing Data in Objects

This is the start of the lectures about data access and operations.

In this and the following sections we will discuss the various operations in classes, in particular how to access data which is encapsulated in objects. By data access we mean both reading (getting) and writing (setting).

In this material we use the word *operation* as a conceptual term. There is nothing called an "operations" in C#. Rather, there are methods, properties, indexers, operators etc. Thus, when we in this teaching material use the word operation it covers - in a broad sense - methods, properties, indexers, operators, events, delegates, and lambda expressions.

17.1. Indirect data access

Lecture 5 - slide 2

It is not a good idea to access the instance variables of a class directly from client classes. We have already discussed this issue in relatively great details in Section 11.3 and Section 11.5.

Data encapsulated in a class is not accessed **directly** from other classes

Rather, data is accessed **indirectly** via operations

So the issue is indirect data access instead of direct data access, when we work on a class from the outside (from other classes, the client classes). When we use indirect data access, the data are accessed through some procedure or function. This procedure or function serves as the *indirection* in between the client, which makes use of the data, and the actual data in the class. This "place of indirection" allows us to carry out checks and other actions in the slipstream of data access. In addition, the procedures and functions that serve as indirection, makes it possible to program certain compensations if the data representation is modified at a later point in time. With this, the client classes of a class C are more likely to survive future modifications of the data, which C encapsulates. It is possible to calculate the data instead of accessing it from variables in the memory.

The following summarizes why indirect data access is better than direct data access.

- Protects and shields the data
  - Possible to check certain conditions each time the data is accessed
  - Possible to carry out certain actions each time the data is accessed
- Makes it easier - in the future - to change the data representation
  - Via "compensations" programmed in the accessing operations
- Makes it possible to avoid the allocation of storage for some data
  - Calculating instead of storing

Protection and shielding may cause the data to be accessed in a conditional data structure. One particular shielding is provided by the precondition of an operation. If the condition does not hold, we may choose not to access the data. In this material, we discuss preconditions in the context of contracts in Chapter 50.

In some circumstances it may be convenient to carry out some action whenever the data of a class is accessed. This is probably most relevant when the data in the class is mutated (assigned to new values). Data values, for which we activate a procedure upon data access, are sometimes called *active values*.

Accessor compensation as a remedy of changing the representation of data is undoubtedly the most important issue. We illustrated this issue in the modifications of Program 11.2, as suggested in Exercise 3.3. In a nutshell, we can often fix the consequences of a shift in data representation by modifying the internals of the operations. By keeping the class interface unchanged all the direct and indirect clients of the affected class will survive. No changes are needed in the client classes. This is the effect of firewalls, as already discussed in Section 11.5.

17.2. Overview of data access in C#

Lecture 5 - slide 3

Below we summarize the various kinds of data access, as supported by operations in C#:

- Directly via public instance variables
    - **Never do that!**
- Indirectly via properties
    - Clients cannot tell the difference between access via properties and direct access of instance variables
- Indirectly via methods
    - Should be reserved for "calculations on objects"
- Indirectly via indexers
    - Provides access by means of the notation known from traditional array indexing
- Indirectly via overloaded operators
    - Provides access by means of the language-defined operator symbols

In the next chapter we will discuss properties in C#. Chapter 19 is about indexers. Methods will be discussed next in Chapter 20. Overloaded operators are treated in Chapter 21.

18. Properties

When a client of a class C accesses data in C via properties, the client of C may have the illusion that it accesses data directly. From a notational point of view, the client of C cannot tell the difference between access to a variable in C and access via a property.

Properties have not been invented in the process of creating C#. Properties have, in some forms, been used in Visual Basic and Delphi (which is a language in the Pascal family). Properties, in the sense discussed below, are not present in Java or C++. Java only allows data to be accessed directly or via methods. Therefore, in Java, it is always possible for clients of a class C to tell if data is accessed directly from a variable in C or indirectly via a method i C. In C#, it is not.

In this material we classify properties as operations, side by side with methods and similar abstractions. Underneath - in the Common Intermediate Language - properties are in fact treated as (getter and setter) methods.

18.1. Properties in C#

Lecture 5 - slide 7

When we use a property it looks like direct access of a variable. But it is not. A variable references to a stored location. A property activates a calculation which is encapsulated in an *abstraction*. The calculations that access data in a class C via properties should be efficient. If not, the clients of C are easily misled. Complicated, time consuming operations should be implemented in methods, see Chapter 20.

Let us first present a very simple, but at the same time a very typical example of properties. In Program 18.1 the `Balance` property accesses the private instance variable `balance`. Notice that the name of the property is capitalized, and that the name of the instance variable is not. This is a widespread convention in many coding styles.

```
using System;

public class BankAccount {

   private string owner;
   private decimal balance;

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

   public decimal Balance {
     get {return balance;}
   }

   public void Deposit(decimal amount){
     balance += amount;
   }

   public void Withdraw(decimal amount){
     balance -= amount;
   }

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

Program 18.1 *A BankAccount class with a trivial Balance property together with Deposit and Withdraw methods.*

In class `BankAccount` it is natural to read the balance via a property, but is problematic to write the balance via a property. Therefore, there is no setter in the `Balance` property. Instead, `Deposit` and `Withdraw` operations (methods) are used. In general we should carefully consider the need for readability and writablity of individual instance variables.

The public `Balance` property as programmed in Program 18.1 provides for read-access to the private instance `balance` variable. You may complain that this is a complicated way of making the instance variable public. What is important, however, is that at a later point in the program evolution process we may change the private data representation. We may, for instance, eliminate the instance variable `balance` entirely, but keep the interface to clients - the `Balance` property - intact. This is illustrated in Program 18.2 below.

```
using System;

public class BankAccount {

   private string owner;
   private decimal[] contributions;
   private int nextContribution;

   public BankAccount(string owner, decimal balance) {
      this.owner = owner;
      contributions = new decimal[100];
      contributions[0] = balance;
      nextContribution = 1;
   }

   public decimal Balance {
     get {decimal result = 0;
          foreach(decimal ctr in contributions)
             result += ctr;
          return result;
         }
   }

   public void Deposit(Decimal amount){
     contributions[nextContribution] = amount;
     nextContribution++;
   }

   public void Withdraw(Decimal amount){
     contributions[nextContribution] = -amount;
     nextContribution++;
   }

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

Program 18.2 *A BankAccount class with a Balance property - without a balance instance variable.*

The interesting thing to notice is that the balance of the bank account now is represented by the decimal array called `contributions` in line 6 of Program 18.2. The `Balance` property in line 16-22 accumulates the contributions in order to calculate the balance of the account.

From a client point of view we can still read the balance of the bank account via the `Balance` property. Underneath, however, the implementation of the `Balance` getter in line 16-22 of Program 18.2 has changed a lot compared to line 14 of Program 18.1. We show a simple client program in Program 18.3, and its output in Listing 18.4 (only on web).

The client program in Program 18.3 can both be used together with `BankAccount` in Program 18.1 and Program 18.2. Thus, the client program has no awareness of the different representation of the balance in the two versions of class `BankAccount`. The only thing that matters in the relation between class `BankAccount` and its client class is the client interface of class `BankAccount`.

```
using System;

class C{

  public static void Main(){
    BankAccount ba = new BankAccount("Peter", 1000);
    Console.WriteLine(ba);

    ba.Deposit(100);
    Console.WriteLine("Balance: {0}", ba.Balance);

    ba.Withdraw(300);
    Console.WriteLine("Balance: {0}", ba.Balance);

    ba.Deposit(100);
    Console.WriteLine("Balance: {0}", ba.Balance);
  }

}
```

Program 18.3   *A client program.*

Above we have discussed *getting of instance variables* in objects. In the `BankAccount` class we have seen how to access to the `balance` instance variable via a getter in the property `Balance`. *Technically*, it is also possible to change the value of the balance instance variable by a *setter*. *Conceptually*, we would rather prefer to update the bank accounts by use of the methods `Deposit` and `WithDraw`. Nevertheless, here is the `Balance` property with both a getter and a setter.

```
public decimal Balance {
  get {return balance;}
  set {balance = value;}
}
```

The setter is activated in an assignment like `b.Balance =` *expression*`;` The important thing to notice is that the property `Balance` is located at the left-hand side of the assignment operator. The value of *expression* is bound to the pseudo variable `value` in the property, and as it appears in the setter, the value of `value` is assigned to the instance variable `balance`.

Properties can also be used for getting and setting fields of struct values. In addition, properties can be used to get and set static variables in both classes and structs.

This ends the essential examples of the `Balance` property of class `BankAccount`. In the web version of the material we provide yet another variation of the `Balance` property in class `BankAccount`. In this example, we enforce a *strict alternation between getting and setting* the balance of a bank account. Please consult the web edition for details.

This ends our discussion of the exotic variation of class `BankAccount`.

Exercise 5.1. *A funny BankAccount*

In this exercises we provide a version of class `BankAccount` with a "funny version" of the `Balance` property. You should access the exercise via the web version, in order to get access to the source programs involved.

Study the `Balance` property of the funny version of class `BankAccount`.

Explain the behaviour of the given client of the funny BankAccount.

Next test-run the program and confirm your understanding of the two classes.

Please notice how difficult it is to follow the details when properties - like `Balance` in the given version of class `BankAccount` - do not pass data directly to and from the instance variables.

18.2. Properties: Class Point with polar coordinates

Lecture 5 - slide 8

We will now look at another very realistic example of properties in C#.

The example in Program 18.8 is a continuation of the `Point` examples in Program 11.2. Originally, and for illustrative purposes, in Program 11.2 we programmed a simple point with public access to its x and y coordinates. We never do that again. The x and y coordinates used in Program 18.8 are called *rectangular coordinates* because they delineate a rectangle between the point and (0,0).

In the class `Point` in Program 18.8 we have changed the data representation to *polar coordinates*. In the paper version of the material we show only selected parts of the class. (We have, in particular, eliminated a set of static methods that convert between rectangular and polar coordinates). In the web version the full class definition is included. Using polar coordinates, a point is represented as a radius and an angle. In addition, and as emphasized with **purple** in Program 18.8 we have programmed four properties, which access the polar and the rectangular coordinates of a point. The properties `Angle` and `Radius` are, of course, trivial, because they just access the underlying private instance variables. The properties `X` and `Y` require some coordinate transformations. We have programmed all the necessary coordinate transformations in static private methods of class `Point`. In the web edition of the material these methods are shown at the bottom of class `Point`.

```
// A versatile version of class Point with Rotation and internal methods
// for rectangular and polar coordinates.

using System;

```

```
public class Point {

  public enum PointRepresentation {Polar, Rectangular}

  private double r, a;            // Polar data representation

  public Point(double x, double y){
     r = RadiusGivenXy(x,y);
     a = AngleGivenXy(x,y);
  }

  public double X {
    get {return XGivenRadiusAngle(r,a);}
  }

  public double Y {
    get {return YGivenRadiusAngle(r,a);}
  }


  public double Radius {
    get {return r;}
  }

  public double Angle{
    get {return a;}
  }

  // Some constructors and methods are not shown

}
```

Program 18.8 *Class Point with polar data representation.*

Exercise 5.2. *Point setters*

In the `Point` class on the accompanying slide we have shown how to program getter properties in the class `Point`. Extend the four properties with setters as well. The new version of this class will support mutable points.

Write a small client program that demonstrates the use of the setter properties.

Hint: Please be aware that the revised class should allow us to get and set the rectangular and polar coordinates (x, y, angle, radius) of a point independent of each other. You should first consider what it means to do so.

18.3. Automatic Properties

Lecture 5 - slide 9

Many of the properties that we write are trivial in the sense that they just get or set a single instance variable. It is tedious to write such trivial properties. It may be possible to ask the programming environment to help with creation of trivial properties. Alternatively in C# 3.0, it is possible for the compiler to generate the trivial properties automatically from short descriptions.

Let us study an example, which extends the initial bank account example from Program 18.1. The example is shown in Program 18.9 and the translation done by the compiler is shown in Program 18.10.

```
1  using System;
2  
3  public class BankAccount{
4  
5    // automatic generation of private instance variables
6  
7    public BankAccount(string owner, decimal balance){
8      this.Owner = owner;
9      this.Balance = balance;
10   }
11 
12   public string Owner {get; set;}
13 
14   public decimal Balance {get; set;}
15 
16   public override string ToString(){
17     return  Owner  + "'s account holds " +  Balance  + " kroner";
18   }
19 }
```

Program 18.9 *Class BankAccount with two automatic properties.*

Based on the formulations in line 12 and 14, the compiler generates the "real properties" shown below in line 13-21 of Program 18.10.

As an additional and important observation, it is no longer necessary to define the private instance variables. The compiler generates the private "backing" instance variables automatically. In terms of the example, the lines 5-6 in Program 18.10 are generated automatically.

As a consequence of automatically generated instance variables, the instance variables cannot be accessed in the class. The names of the instance variables are unknown, and therefore they cannot be used at all! Instead, the programmer of the class accesses the hidden instance variables through the properties. As an example, the owner and balance are accessed via the properties `Owner` and `Balance` in line 17 of the `ToString` method in Program 18.9.

```
1  using System;
2  
3  public class BankAccount{
4  
5    private string _owner;
6    private decimal _balance;
7  
8    public BankAccount(string owner, decimal balance){
9      _owner = owner;
10     _balance = balance;
11   }
12 
13   public string Owner {
14     get {return _owner;}
15     set {_owner = value;}
16   }
17 
18   public decimal Balance {
19     get {return _balance;}
20     set {_balance = value;}
21   }
```

```

  public override string ToString(){
    return  Owner  + "'s account holds " +  Balance  + " kroner";
  }
}
```

Program 18.10   *An equivalent BankAccount class without automatic properties.*

Because properties, internally in a class, play the role of instance variables, it is hardly meaningful to have an automatic property with a getter and no setter. The reason is that the underlying instance variable cannot be initialized - simple because its name is not available. Similarly, an automatic property with a setter, but not getter, would not make sense, because we would not be able to access the underlying instance variable. Therefore the compiler enforces that an automatic property has both a getter and a setter. It is possible, however, to limit the visibility of either the getter or setter (but not both). As an example, the following definition of the `Balance` property

```
   public string Balance {get; private set;}
```

provides for public reading, but only writing from within the class. In class `BankAccount`, this is probably what we want. It is OK to access the balance from outside, but we the account to be updated by use of either the `Deposit` method or the `Withdraw` method.

Finally, let us observe that the syntax of automatic properties is similar to the syntax used for abstract properties, see Section 30.3.

Automatic properties: Reflections and recommendations. FOCUS BOX 18.1

I recommend that you use automatic properties with some care! It is worth emphasizing that a class almost always encapsulates some state - instance variables - some of which can be accessed by properties. It feels strange that we can program in a way - with automatic properties - without ever declaring the instance variables explicitly. And it feels strange that we never, from within the class, refers to the instance variables.

Automatic properties are useful in the early life of a class, where we allows for direct (one-to-one) access to the instance variables via methods or properties. In an early version of the class `Point` of Section 11.6, this was the case. (We actually started with public instance variables, but this mistake is taken care of in Exercise 3.3).

Later on, we may internally invent a more sophisticated data representation - or we may change our mind with respect to the data representation. In the `Point` class we went from a rectangular representation to a polar representation. The data representation details should always be a private and internal concern of the class. When this change happens we have to introduce instance variables, exactly as described in Section 11.8. When doing so we have to get rid of the automatic properties. The introduction of instance variables and the substitution of the automatic properties with 'real properties' represent internal changes to the `Point` class. The clients of `Point` will not be affected by these changes. *This is the point!* In the rest of the lifetime of class `Point`, it is unlikely that automatic properties will be 'part of the story'.

Automatic properties contribute to a C# 3.0 *convenience layer* on top of already existing means of expressions.

18.4. Object Initialization via Properties

Lecture 5 - slide 10

In this section we will see that property setters in C#3.0 can be used for initialization purposes in the slipstream of constructors. The properties that we use for such purposes can be automatic properties, as discussed in Section 18.3.

Let us again look at a simple `BankAccount` class, see Program 18.11. The class has two automatic properties, backed by two instance variables, which we cannot access. In addition, the class has two constructors, cf. Section 12.4.

```
using System;

public class BankAccount{

  // automatic generation of private instance variables

  public BankAccount(): this("NN") {
  }

  public BankAccount(string owner){
    this.Balance = 0;
    this.Owner = owner;
  }

  public string Owner {get; set;}

  public decimal Balance {get; set;}

  public override string ToString(){
    return Owner + "'s account holds " + Balance + " kroner";
  }
}
```

Program 18.11 *Class BankAccount - with two constructors.*

Below, in Program 18.12 we make an instance of class `BankAccount` in line 6. As emphasized in **purple** the owner and balance is initialized by a so-called *object initializer*, in curly brackets, right after `new BankAccount`. The initializer refers the setter of the automatic properties `Owner` and `Balance`. All together we have made a new bank account by use of the parameterless constructor. The initialization is done by the setters of the `Owner` and the `Balance` properties.

In line 7 of Program 18.12 we make another instance of class `BankAccount`, where `Owner` is initialized via the actual parameter "Bill" passed to the constructor, and the balance is initialized via an object initializer in curly brackets.

```
1  using System;
2  
3  public class Client {
4  
5    public static void Main(){
6      BankAccount ba1 = new BankAccount{Owner = "James", Balance = 250},
7                  ba2 = new BankAccount("Bill"){Balance = 1200};
8  
9  
10     Console.WriteLine(ba1);
11     Console.WriteLine(ba2);
12   }
13 
14 }
```

Program 18.12 *A client of class BankAccount with an object initializer.*

The compiler translates line 6 of Program 18.12 to line 6-8 in Program 18.13. Similarly, line 7 of Program 18.12 is translated to line 10-11 in Program 18.13.

```
1  using System;
2  
3  public class Client {
4  
5    public static void Main(){
6      BankAccount ba1 = new BankAccount();
7      ba1.Owner = "James";
8      ba1.Balance = 250;
9  
10     BankAccount ba2 = new BankAccount("Bill");
11     ba2.Balance = 1200;
12 
13     Console.WriteLine(ba1);
14     Console.WriteLine(ba2);
15   }
16 
17 }
```

Program 18.13 *An equivalent client of class BankAccount without object initializers.*

Let us, finally, elaborate the example with the purpose of demonstrating *nested object initializers*. In Program 18.14 we show the classes `BankAccount` and `Person`. As can be seen, an instance of class `BankAccount` refers to an owner which is an instance of class `Person`. It turns out to be crucial for the example that the `Owner` of a `BankAccount` refers to a `Person` object (more specifically, that it is not a `null` reference).

```
using System;

public class BankAccount{

  public BankAccount(){
     this.Balance = 0;
     this.Owner = new Person();
  }

  public Person Owner {get; set;}
  public decimal Balance {get; set;}
12
13   public override string ToString(){
14     return Owner + "'s account holds " + Balance + " kroner";
15   }
16 }
17
18 public class Person {
19   public string FirstName {get; set;}
20   public string LastName {get; set;}
21
22   public override string ToString(){
23     return FirstName + " " + LastName;
24   }
25 }
```

Program 18.14 *Class BankAccount and class Person.*

In the `Client` class, see Program 18.15 we make two `BankAccounts` , `ba1` and `ba2`. The initializers, emphasized in line 7-9 and 12-14, initialize the `Owner` of a `BankAccount` with *nested object initializers*. Notice that there is no new operator in front of `{FirstName = ..., LastName = ...}`. The `Person` object already exists. In this example, it is instantiated in line 7 of Program 18.14 together with the `BankAccount`.

```
1  using System;
2
3  public class Client {
4
5    public static void Main(){
6      BankAccount ba1 = new BankAccount{
7                              Owner = {FirstName = "James",
8                                       LastName = "Madsen"},
9                              Balance = 250},
10
11                 ba2 = new BankAccount{
12                              Owner = {FirstName = "Bill",
13                                       LastName = "Jensen"},
14                              Balance = 500};
15
16     Console.WriteLine(ba1);
17     Console.WriteLine(ba2);
18   }
19
20 }
```

Program 18.15 *A client of class BankAccount with nested object initializers.*

The use of property names (setters) in object initializers gives the effect of *keyword parameters*. Keyword parameters refer to a formal parameter name in the actual parameter list. Keyword parameters can be given in any order, as a contrast to *positional parameters*, which must given in the order dictated by the formal parameter list. In addition, the caller of an abstraction that accepts keyword parameters can choose not to pass certain keyword parameters. In that case, defaults will apply. In case of C#, such default values can be defined within the body of the constructors.

18.5. Summary of properties in C#

Lecture 5 - slide 11

The syntax of property declarations is shown in Syntax 18.1. Both the get part and the set part are optional. Syntactically, the signature a property declaration is like a method declaration without formal parameters. As can be seen, the syntax of a property body is very different from the syntax of a method body.

```
 modifiers return-type property-name{
  get {body-of-get}
  set {body-of-set}
}
```

Syntax 18.1 *The syntax of a C# property. Here we show both the getter and setter. At least one of them must be present.*

The following summarizes the characteristics of properties in C#:

- Provides for either read-only, write-only, or read-write access
- Both instance and class (static) properties make sense
- Property setting appears on the left of an assignment, and in `++` and `--`
- Trivial properties can be defined "automatically"
- Properties should be fast and without unnecessary side-effects

The following observations about property naming reflect a given coding style. A coding style is not enforced by the compiler.

A C# property will often have the same name as a private data member

The name of the property is capitalized - the name of the data member is not

19. Indexers

From a notational point of view it is often attractive to access the data, encapsulated in a class or struct, via conventional array notation. Indexers are targeted to provide array notation on class instances and struct values.

Indexers can be understood as a specialized kind of properties, see Chapter 18. Both indexers and properties are classified as *operations* in this material, together with methods and other similar abstractions.

19.1. Indexers in C#

Lecture 5 - slide 13

Indexers allow access to data in an object with use of array notation

The important benefit of indexers is the notation they make available to their clients.

Let us assume that **`v`** is a variable that holds a reference to an object *obj*. With use of methods we can access data in *obj* with **`v.method(parameters)`**. In Chapter 18 we introduced properties and the property access notation **`v.property`**. We will now introduce the notation **`v[i]`**, where **`i`** typically (but not necessarily) is some integer (index) value.

We will start with an artificial ABC example in Program 19.1 which tells how to define an indexer in a class that encapsulates three instance variables `d`, `e`, and `f`. The indexer is used in a client class in Program 19.2. The example introduces the indexer abstraction, but it is not a typical use of an indexer.

As it can be seen in Program 19.1 an indexer must be named "this". Like a property, it has a getter and a setter part.

The getter is activated when we encounter an expression `a[i]`, where `a` is a variable of type `A`. The body of the getter determines the value of `a[i]`.

The setter is activated when we encounter an assignment like **`a[i]`** = *expression*. The value of *expression* is bound to the implicit parameter named `value`. The body of the setter determines the effect on the instance variables in *obj* upon execution of `a[i]` = *expression*.

In Program 19.1 `a[1]` accesses the instance variable `d`, `a[2]` accesses the instance variable `e`, and a[3] accesses the instance variable `f`. This is not a typical arrangement, however. Most often, indexers are used to access members of a data collection.

```
using System;

public class A {
  private double d, e, f;

  public A(double v){
    d = e = f = v;
  }

  public double this [int i]{
   get {
     switch (i){
       case 1: {return d;}
       case 2: {return e;}
       case 3: {return f;}
       default: throw new Exception("Error");
     }
   }
   set {
     switch (i){
       case 1: {d = value; break;}
       case 2: {e = value; break;}
       case 3: {f = value; break;}
       default: throw new Exception("Error");
    }
   }
  }

  public override string ToString(){
    return "A: " + d + ", " + e + ", " + f;
  }

}
```

Program 19.1 *A Class A with an indexer.*

Program 19.2 shows the indexer from Program 19.1 in action. First, in line 9, we illustrate the three setters, where `a[i]` occurs at the left-hand side of the assignment symbol. Following that, in line 11, we illustrate two getters. The output of Program 19.2 is shown in Listing 19.3 (only on web).

```
1  using System;
2  
3  class B {
4  
5    public static void Main(){
6      A a = new A(5);
7      double d;
8  
9      a[1] = 6; a[2] = 7.0; a[3] = 8.0;
10 
11     d = a[1] + a[2];
12 
13 
14     Console.WriteLine("a: {0}, d: {1}", a, d);
15   }
16 }
```

Program 19.2 *A client of A which uses the indexer of A.*

As an additional example of indexers we will study the class `BitArray`. This example is only present in the web-version of the material.

19.2. Associative Arrays

Lecture 5 - slide 14

In Section 19.1 we showed how to index an object with a single integer index. In this section we will demonstrate that the indexing value can have an arbitrary type. Thus, the type of `obj` in `a[obj]` can be an arbitrary type in C#, for instance a type we program ourselves.

An *associative array* is an array which allows indexing by means of arbitrary objects, not just integers

An associative arrays maps a set of objects (the indexing objects, keys) to another set of objects (the element objects).

In Program 19.6 we illustrate how to index instances of class A with strings instead of integers. If you understood Program 19.1 and Program 19.2 it will also be easy to understand Program 19.6 and Program 19.7. Notice in this context that C#, very conveniently, allows switch control structures to switch on strings. The program output is shown in Listing 19.8 (only on web).

```
using System;

public class A {
  private double d, e, f;

  public A(double v){
    d = e = f = v;
  }

  public double this [string str]{
   get {
     switch (str){
       case "d": {return d;}
       case "e": {return e;}
       case "f": {return f;}
       default: throw new Exception("Error");
     }
   }
   set {
     switch (str){
       case "d": {d = value; break;}
       case "e": {e = value; break;}
       case "f": {f = value; break;}
       default: throw new Exception("Error");
    }
   }
  }

  public override string ToString(){
    return "A: " + d + ", " + e + ", " + f;
  }

}
```

Program 19.6 *The class A indexed by a string.*

```
using System;

class B {

  public static void Main(){
    A a = new A(5);
    double d;

    a["d"] = 6; a["e"] = 7.0; a["f"] = 8.0;

    d = a["e"] + a["f"];   // corresponds to d = a.d + a.e
                           // in case d and e had been public

    Console.WriteLine("a: {0}, d: {1}", a, d);
  }
}
```

Program 19.7 *A client of A which uses the string indexing of A.*

We have seen that it makes sense to index with strings, and more generally with an arbitrary instance of a class. In fact, it is possible to base the indexing on two or more objects. This is, of course, important if we index multi-dimensional data structures.

Associative arrays are in C# implemented by means of hashtables in dictionaries

In the lecture about collections, see Chapter 46, we will see how to make use so-called dictionaries (typically implemented as hash tables) for efficient data structures that map a set of objects to another set of objects. Indexers, as discussed in this section chapter, provide a convenient surface notation to deal with such dictionaries. In Section 46.2 the indexer prescribed by the generic interface **`IDictionary<K,V>`** accesses objects of type `V` via an index of type `K`.

19.3. Summary of indexers in C#

Lecture 5 - slide 15

Here follows a syntax diagram of indexers:

```
 modifiers return-type this[formal-parameter-list]
  get {body-of-get}
  set {body-of-set}
}
```

Syntax 19.1 *The syntax of a C# indexer*

It is similar to the syntax diagram of properties, as shown in Syntax 18.1

The main characteristics of indexers are as follows:

- Provide for indexed read-only, write-only, or read-write access to data in objects
- Indexers can only be instance members - not static
- The indexing can be based on one, two or more formal parameters
- Indexers can be overloaded
- Indexers should be without unnecessary side-effects
- Indexers should be fast

20. Methods

Methods are the most important kind of operations in C#. Methods are more fundamental than properties and indexers. We would be able to do object-oriented programming without properties and indexers (by implementing all properties and indexers as methods), but not without methods. In Java, for instance, there are methods but no properties and no indexers.

A method is a procedure or a function, which is part of a class. Methods (are supposed to) access (operate on) the data, which are encapsulated by the class. Methods should be devoted to nontrivial operations. Trivial operations that just read or write individual instance variables should in C# be programmed as properties.

We have already in Section 11.9 and Section 11.11 studied the fundamentals of methods in an object-oriented programming language. In these sections we made the distinction between instance methods and class methods. Stated briefly, instance methods operate on instance variables (and perhaps class variables as well). Class methods (called static methods in C#) can only operate on class variables (static variables in C#).

When we do object-oriented programming we organize most data in instances of classes (objects) and in values of struct types. We only use class related data (in static variables) to a lesser degree. Therefore instance methods are more important to us than class methods. In the rest of this chapter we will therefore focus on instance methods.

This chapter is long because we have to cover a number of different parameter passing modes. If you only need a basic understanding of methods and the most frequently used parameter passing mode - call-by-value parameters - you should read until (and including) Section 20.4 and afterwards proceed to Chapter 21.

20.1. Local variables in methods

Lecture 5 - slide 18

Local variables of a method are declared in the *statements* part of the block relative to Syntax 20.1. Local variables are short lived; They only exists during the activation of the method.

```
modifiers return-type method-name(formal-parameter-list){
  statements
}
```

Syntax 20.1 *The syntax of a method in C#*

You should notice the difference between *local variables* and *parameters*, which we discuss below in Section 20.2. Parameters are passed and initialized when the method is activated. The initial value comes from an actual parameter. Local variables are introduced in the *statements* part (the body) of the method, and as explained below they may - or may not - be initialized explicitly.

You should also notice the difference between *local variables* and *instance variables* of a class. A local variable only exists in a single call of the method. An instance variable exists during the lifetime of an object.

Local variables

- May be declared anywhere in the block
    - Not necessarily in the initial part of the block
- Can be initialized by an initializer
- Can alternatively be declared without an initializer
    - No default value is assigned to a local variable
        - Different from instance and class variables which are given default values
    - The compiler will complain if a local variable is referred without prior assignment

In the program below we contrast instance variables of the class `InitDemo` with local variables in the method `Operation` of `InitDemo`. The **purple** instance variables are implicitly initialized to their default values. The **blue** local variables in `Operations` are not. The program does not compile. In line 18 and 19 the compiler will complain about use of unassigned local variables.

```
1  using System;
2  
3  class InitDemo{
4  
5    private int intInstanceVar;
6    private bool boolInstanceVar;
7  
8    public void Operation(){
9      int intLocalVar;
10     bool boolLocalVar;
11 
12     Console.WriteLine("intInstanceVar: {0}. boolInstanceVar: {1}",
13                       intInstanceVar,
14                       boolInstanceVar);
15 
16     // Compile time errors:
17     Console.WriteLine("intLocalVar: {0}. boolLocalVar: {1}",
18                       intLocalVar,
19                       boolLocalVar);
20 
21   }
22 
23   public static void Main(){
24     new InitDemo().Operation();
25   }
26 
27 }
```

Program 20.1 *Local variables and instance variables - initialization and default values.*

In C#, Java and similar languages there is no such thing as a *global variable*. This is often a problem for programmers who are used to pass data around via global variables. If you really, really need a global variable in C#, the best option is to use a class (static) variable in one of your top-level classes. In general, however, it is a better alternative to pass data around via parameters (such as parameters to constructors).

20.2. Parameters

Lecture 5 - slide 19

As a natural counterpart to Syntax 18.1 (of properties) and Syntax 19.1 (of indexers) we have shown the syntactical form of a method in Syntax 20.1. The syntactical characteristic of methods, in contrast to properties and indexers, is the formal parameters in ordinary, soft parentheses: `(...)`. Even a method with no parameters must have an empty pair of parentheses ( ) - in both the method definition and in the method activation. Properties have no formal parameters, and indexers have formal parameters in brackets: `[...]`.

We will now discuss parameter passing in general. The following introduces *formal parameters* and *actual parameters*.

Actual parameters occur in a call. *Formal parameters* occur in the method declaration. In general, actual parameters are expressions which are evaluated to *arguments*. Depending on the kind of parameters, the arguments are somehow associated with the formal parameters.

C# offers several different parameter passing modes:

- Value parameters
  - The default parameter passing mode, without use of a modifier
- Reference parameters
  - Specified with the **`ref`** modifier
- Output parameters
  - Specified with the **`out`** modifier
- Parameter arrays
  - Value parameters specified with the **`params`** modifier

Far the majority of the parameters in a C# program are passed by value. Thus, the use of value parameters is the most important parameter passing technique for you to understand. Value parameters are discussed in Section 20.3 - Section 20.5.

Reference and output parameters are closely related to each other, and they are only rarely used in object-oriented C# programs. Output parameter can be seen as a restricted version of reference parameters. Reference parameters stem from *variable parameters* (`var` parameters) in Pascal. Reference parameters are discussed in Section 20.6 and out parameters are discussed in Section 20.7.

Parameter arrays cover the idea that a number of actual value parameters (of the same type) are collected into an array. In this way, parameter arrays provide for a more sophisticated correspondence between the arguments and the formal parameters. C# parameter arrays are discussed in Section 20.9.

It is, in general, an issue how a given actual parameter (or argument) is related to a formal parameter. This is called *parameter correspondence*. With value, **`ref`** and **`out`** parameter we use *positional parameter correspondence*. This is simple. The first formal parameter is related to the first actual parameter, the second to the second, etc. With parameter arrays, a number of actual parameters - all remaining actual parameters - correspond to a single formal parameter.

In general, there are other parameter correspondences, most notable *keyword parameters*, where the name of the formal parameter is used together with the actual parameter. Keyword parameters are not used directly in C#. But as we have seen in Section 18.4 a kind of keyword parameters is used when properties are used for object initialization in the context of the `new` operator.

In Section 6.9 - Program 6.20 - we have shown a program example that illustrate multiple parameter passing modes in C#. If you wish the ultra short description of parameter passing in C# you can read Section 6.9 instead of Section 20.3 - Section 20.9.

20.3. Value Parameters

Lecture 5 - slide 20

Value parameters are used for input to methods

When the story should told briefly, *call-by-value parameters* (or just *value parameters*) are used for input to methods. The output from a method is handled via the value returned from the method - via use of **`return`**. This simple version of the story is true for the majority of the methods we write in our classes. But as we will see already in Section 20.4 it is also possible to handle some kinds of output via references to objects passed as value parameters.

Value parameter passing works in the following way:

- A formal parameter corresponds to a local variable
- A formal parameter is initialized by the corresponding argument (the value of the actual parameter expression)
    - A *copy* of the argument is bound to the formal parameter
- Implicit conversions may take place
- A formal parameter is assignable, but with no effect outside the method

We have already seen many examples of value parameters. In case you want to review typical examples, please consult the `Move` method of class `Point` in Program 11.2, the `Withdraw` and `Deposit` methods of class `BankAccount` in Program 11.8, and the `AgeAsOf` method in class `Person` in Program 16.10.

20.4. Passing references as value parameters

Lecture 5 - slide 21

Care must be taken if we pass references as value parameters

Most of the data we deal with in object-oriented C# programs are represented as instances of classes - as objects. This implies that such data are accessed by references. Again and again we pass such references as value parameters to methods. Therefore we must understand - in details - what happens.

Here is the short version of the story. Let us assume that send the message `DayDifference` to a `Date` object with another `Date` object as parameter:

```
someDate.DayDifference(otherDate)
```

`Date` is a class, and therefore both `someDate` and `otherDate` hold references to `Date` objects. It is possible for the `DayDifference` method to mutate the `Date` object referred by `otherDate`, despite the fact that the parameter of `DayDifference` is a value parameter. It is not, however, possible for `DayDifference` to modify value of the variable (actual parameter) `otherDate` as such.

In the web-version we present the source program behind the example is great details.

The insight obtained in this section is summarized as follows.

> In case a reference is passed as an argument to a value parameter, the referenced object can be modified through the formal parameter

---

Exercise 5.3. *Passing references as ref parameters*

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 `Date` and `Person` classes of the corresponding slide we pass a reference as *a value parameter* to method `DayDifference` in class `Date`. Be sure to understand this. Read about **ref** parameters later in this lecture.

Assume in this exercise that the formal parameter `other` in `Date.DayDifference` is passed by reference (as a C# **ref** parameter). Similarly, the actual parameter `dateOfBirth` to `DayDifference` should (of course) be passed by reference (using the keyword `ref` in front of the actual parameter).

What will be the difference caused by this program modification.

Test-drive the program with a suitable client to verify your answer.

20.5. Passing structs as value parameters

Lecture 5 - slide 22

This section is parallel to Section 20.4. In this section we pass a struct value (as opposed to an instance of a class) as a value parameter. Our finding is that a struct value (the birthday value of type `Date`) cannot be mutated from the `DayDifference` method of struct `Date`.

If we assume that `Date` is a struct instead of a class, the expression

```
someDate.DayDifference(otherDate)
```

passes a copy of `otherDate` to `DayDifference`. The copy is discarded when we return from `DayDifference`. If `DayDifference` mutates the value of its parameter, only the local copy is affected. The value of `otherDate` is not!

In the web-version we will discuss the same example as in the web-version of Section 20.4.

Notice the following observation.

> There is a good fit between use of value types and call-by-value parameter passing

If you wish the best possible fit (and no surprises) you should use value parameters with value types. The use of struct `Date` instead of class `Date` also alleviates the privacy leak problem, as pointed out in Section 16.5. See also Exercise 4.3.

Exercise 5.4. *Passing struct values as ref parameters*

This exercise corresponds to the similar exercise on the previous slide.

In the `Date` struct and the `Person` class of this slide we pass a struct value as a *value parameter* to the method `DayDifference`.

Assume in this exercise that the formal parameter `other` in `Date.DayDifference` is passed by reference (a C# **ref** parameter). Similarly, the actual parameter `dateOfBirth` to `DayDifference` should (of course) be passed by reference (using the keyword `ref` in front of the actual parameter).

What will be the difference caused by this program modification. You should compare with the version on on the slide.

Test-drive the program with a suitable client to verify your answer.

20.6. Reference Parameters

Lecture 5 - slide 23

In C, call-by-reference parameters are obtained by passing pointers as value parameters. Reference parameters in C# are **not** the same as call-by-reference parameters in C.

Reference parameters in C# are modeled after **var** parameters in Pascal. Stated briefly, a formal reference parameter in C# is an alias of the corresponding actual parameter. Therefore, the actual parameter must be a variable.

Reference parameters can be used for both input to and output from methods

Reference parameters can be used to establish alternative names (aliases) of already existing variables. The alternative names are used as formal parameters. Once established, such parameters can be used for both input and output purposes relative to a method call. The established aliases exist until the method returns to its caller.

If we - in C# - only are interested in using reference parameters for output purposes we should use **out** parameters, see Section 20.7.

Reference parameters work in the following way:

- The corresponding argument must be a variable, and it must have a value
- The types of the formal parameter and the argument must be identical
- The formal parameter becomes another name (an alias) of the argument

- The actual parameter must be prefixed with the keyword **`ref`**

In the first item it is stated that an actual reference parameter (which is a variable) must have a value before it is passed. In C#, this is called *definite assignment*.

As described in the fourth item, and as a novel contribution of C#, it is necessary to mark both formal and actual parameter with the **ref** keyword. In most other languages, only the formal parameter is marked. This may seem to be a little detail, but it implies that it is easy to spot reference parameters in a method calling form. This is very useful.

We show an example of reference parameters in Program 20.10: Swapping the values of two variables. This is the example used over and over, when reference parameters are explained.

```
using System;

public class A{
  private int a, b, c;

  public A(){
    a = 1; b = 2; c = 3;
  }

  public void Swap(ref int v1, ref int v2){
    int temp;
    temp = v1; v1 = v2; v2 = temp;
  }

  public override string ToString(){
    return String.Format("{0} {1} {2}", a, b, c);
  }

  public void Go(){
    Console.WriteLine("{0}", this);
    Swap(ref a, ref b); Swap(ref b, ref c);
    Console.WriteLine("{0}", this);
  }

  public static void Main(){
    new A().Go();
  }
}
```

Program 20.10 *The class A with a Swap method.*

In Program 20.10 we instantiate the class itself (class `A`) in the `Main` method. We send the parameterless message `Go` to this object. Hereby we take the transition from a "static situation" to an "object situation". Without this transition it would not have been possible to use the instance variables `a`, `b`, and `c` in class `A`. (They should instead have been static variables). The `Go` method pattern illustrated here is inspired from [Bishop04].

```
1 2 3
2 3 1
```

Listing 20.11 *Output of class A with Swap.*

It seems natural to support more than just value parameters in an ambitious, real-world programming language. But it is worth a consideration how much - and in which situations - to use it. We will discuss this in some details in Section 20.8.

20.7. Output Parameters

Lecture 5 - slide 24

Output parameters in C# are reference parameters used only for output purposes.

Output parameters are used for output from methods. The method is supposed to assign values to output parameters.

Here follows the detailed rules of output parameter passing:

- The corresponding argument must be a variable
- The corresponding argument needs not to have a value on beforehand
- The formal parameter should/must be assigned by the method
- The formal parameter becomes another name (an alias) of the argument
- The types of the formal parameter and the argument must be identical
- The actual parameter must be prefixed with the keyword **`out`**

Notice the second item: It is not necessary that the actual parameter has a value before the call. In fact, the purpose of the **out** parameter is exactly to (re)initialize the actual **out** parameters. The method must ensure that the output parameter has a value (is definitely assigned) when it returns.

In Program 20.12 `DoAdd` returns the sum of the parameters `v1`, `v2`, and `v3` in the last parameter `v`. The corresponding actual parameter `r` is initialized by the call to `DoAdd` in line 21 of Program 20.12. I wrote `DoAdd` to demonstrate output parameters. Had it not been for this purpose, I would have returned the sum from `DoAdd`. In that way `DoAdd` should be called as `result = DoAdd(v1, v2, v3)`. In this case I would de-emphasize the imperative nature by calling it just `Add`. These changes describe a transition from an imperative to a more functional programming style.

```
1  using System;
2  
3  public class A{
4    private int a, b, c;
5    private int r;
6  
7    public A(){
8      a = 1; b = 2; c = 3;
9    }
10 
11   public void DoAdd(int v1, int v2, int v3, out int v){
12     v = v1 + v2 + v3;
13   }
14 
15   public override string ToString(){
16     return String.Format("{0} {1} {2}. {3}", a, b, c, r);
17   }
18 
```

```
  public void Go(){
    Console.WriteLine("{0}", this);
    DoAdd(a, b, c, out r);
    Console.WriteLine("{0}", this);
  }

  public static void Main(){
    new A().Go();
  }
}
```

Program 20.12 *The class A with a DoAdd method.*

```
1 2 3. 0
1 2 3. 6
```

Listing 20.13 *Output of class A with DoAdd.*

In Program 20.14 of Section 20.9 we show a variant of Program 20.12, which allows for an arbitrary number of actual parameters. This variant of the program is introduced with the purpose of illustrating parameter arrays.

20.8. Use of ref and out parameters in OOP

Lecture 5 - slide 25

It is interesting to wonder about the fit between object-oriented programming and the use of reference parameters. Therefore the following question is relevant.

How useful are reference and output parameters in object-oriented programming?

Output parameters are useful in case we program a method which need to produce two or more pieces of output. In such a situation, we face the following possibilities:

- Use a number of **`out`** parameters
- Mutate objects passed by value parameters
- Aggregate the pieces of output in an object and return it
- Assign the output to instance variables which subsequently can be accessed by the caller

Let us first face the first item. If a (public) method needs to pass back (to its caller) more than one piece of output, which are only loosely related to each other, it may be the best solution to use one or more out parameters for these. It should be considered to pass one of the results back via `return`.

In a language with **`ref`** and **`out`** parameters it is confusing to pass results out of method via references passed by value (call-by-value). Use a **`ref`** or an **`out`** parameter!

If the pieces of output are related - if they together form a concept - it may be reasonable to aggregate the pieces of output in a new struct or a new class. An instance of the new type can then be returned via **`return`**.

Assignment of multiple pieces of output to instance variables in the enclosing class, and subsequent access of these via properties, may compromise the original idea of the class to which the method belongs. It may also pollute the class. Therefore, it should be avoided.

`ref` and `out` parameters are relatively rare in the C# standard libraries

In summary, we see that there are several alternatives to the use of reference and output parameters in an object-oriented context.

Reference (`ref`) and output (`out`) parameters are only used in very few methods in the .NET framework libraries. In general, it seems to be the case that reference parameters and output parameters are not central to object-oriented programming.

20.9. Parameter Arrays

Lecture 5 - slide 26

In the previous sections we have discussed various parameter passing techniques. They were all related to the meaning of the formal parameter relative to the corresponding actual parameter (and the argument derived from the actual parameter).

In this section we will concentrate on the *parameter correspondence* mechanism. In the parameter passing discussed until now there has been a one-to-one correspondence between formal parameters and actual parameters. In this section we will study a parameter passing technique where zero, one, or more actual parameters correspond to a single formal parameter.

A parameter array is a formal parameter of array type which can absorb zero, one or more actual parameters

A formal parameter list in C# starts with a number of value/reference/output parameters for which there must exist corresponding actual parameters in a method activation. Following these parameters there can be a single parameter array, which collects all remaining arguments in an array.

With parameter arrays, there can be arbitrary many actual parameters following the 'ordinary parameters', but not arbitrary few. There must always be so many actual parameters that all the 'required' formal parameters (before a possible parameter arrays) are associated.

The following rules pertain to use of parameter arrays in C#:

- Elements of a parameter array are passed by value
- A parameter array must be the last parameter in the formal parameter list
- The formal parameter must be a single dimensional array
- Arguments that follow ordinary value, ref and out parameters are put into a newly allocated array object
- Arguments are implicitly converted to the element type of the array

It is easiest to understand parameter arrays from an example, such as Program 20.14. This is a variant of Program 20.12, which we have already discussed in Section 20.7. The `DoAdd` method takes one required parameter (which is an output parameter). As the actual out parameter in line 24, 27, and 30 we use the instance variable `r`. Following this parameter comes the parameter array called `iv` (for input values). All actual parameters after the first one must be of type `int`, and they are collected and inserted into an integer array, and made available to `DoAdd` as the `int` array `iv`.

In Program 20.14 the `DoAdd` messages to the current object add various combinations of the instance variables `a`, `b`, and `c` together. The result is assigned to the out parameter of `DoAdd`.

```
1  using System;
2  
3  public class A{
4    private int a, b, c;
5    private int r;
6  
7    public A(){
8      a = 1; b = 2; c = 3;
9    }
10 
11   public void DoAdd(out int v, params int[] iv){
12     v = 0;
13     foreach(int i in iv)
14       v += i;
15   }
16 
17   public override string ToString(){
18     return String.Format("{0} {1} {2}. {3}", a, b, c, r);
19   }
20 
21   public void Go(){
22     Console.WriteLine("{0}", this);
23 
24     DoAdd(out r, a, b, c);
25     Console.WriteLine("{0}", this);
26 
27     DoAdd(out r, a, b, c, a, b, c);
28     Console.WriteLine("{0}", this);
29 
30     DoAdd(out r);
31     Console.WriteLine("{0}", this);
32   }
33 
34   public static void Main(){
35     new A().Go();
36   }
37 }
```

Program 20.14 *The class A with a DoAdd method - both out and params.*

The output of Program 20.14 is shown in Listing 20.15. We have emphasized the value of `r` after each `DoAdd` message. Be sure that you are able to understand the parameter passing details.

```
1 2 3. 0
1 2 3. 6
1 2 3. 12
1 2 3. 0
```

Listing 20.15 *Output of class A with DoAdd - both out and params.*

The type constraint on the actual parameters can be 'removed' by having a formal parameter array of type `Object[]`.

When we studied nullable types in Section 14.9 we encountered the `IntSequence` class. In line 5 of Program 14.17 you can see a parameter array of the constructor. Please notice the flexibility it gives in the construction of new `IntSequence` objects. See also Program 14.19 where `IntSequence` is instantiated with use of the mentioned constructor.

20.10. Extension Methods

Lecture 5 - slide 27

In C#3.0 an extension method defines an instance method in an existing class without altering the definition of the class. The use of extension methods is convenient if you do not have access to the source code of the class, in which we want to have a new instance method.

Let us look at an example in order to illustrate the mechanisms behind the class extension. We use Program 11.3 as the starting point. In line 26-31 of Program 11.3 it can be observed that we have inlined calculation of distances between pairs of points. It would embellish the program if we had called `p.DistanceTo(q)` instead of

```
Math.Sqrt((p.x - q.x) * (p.x - q.x) +
          (p.y - q.y) * (p.y - q.y));
```

to find the distance between to points `p` and `q`. We will now assume that the instance method `DistanceTo` can be used on an instance of class `Point`. Program 20.16 below shows the embellishment of Program 11.3.

```
// A client of Point that instantiates three points and calculates
// the circumference of the implied triangle.

using System;

public class Application{

  public static void Main(){

    Point  p1 = PromptPoint("Enter first point"),
           p2 = PromptPoint("Enter second point"),
           p3 = PromptPoint("Enter third point");

    double p1p2Dist = p1.DistanceTo(p2),
           p2p3Dist = p2.DistanceTo(p3),
           p3p1Dist = p3.DistanceTo(p1);

    double circumference = p1p2Dist + p2p3Dist + p3p1Dist;

    Console.WriteLine("Circumference: {0} {1} {2}: {3}",
                       p1, p2, p3, circumference);
  }

  public static Point PromptPoint(string prompt){
    Console.WriteLine(prompt);
    double x = double.Parse(Console.ReadLine()),
           y = double.Parse(Console.ReadLine());
    return new Point(x,y, Point.PointRepresentation.Rectangular);
  }

}
```

Program 20.16 *A Client of class Point which uses an extension method DistanceTo.*

It is possible to extend class `Point` with the instance method `DistanceTo` without altering the source code of class `Point`. In a static class, such as in class `PointExtensions` shown below in Program 20.17, we define the `DistanceTo` method. It is defined as a static method with a first parameter prefixed with the keyword **`this`**. The C#3.0 compiler is able to translate an expression like `q.DistanceTo(q)` to `PointExtensions.DistanceTo(p,q)`. This is the noteworthy "trick" behind extension methods.

```
using System;

public static class PointExtensions{

  public static double DistanceTo(this Point p1, Point p2){
    return Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) +
                     (p1.Y - p2.Y) * (p1.Y - p2.Y));
  }

}
```

Program 20.17 *The static class PointExtensions.*

In summary, an extension method

- extends the type of the first parameter
- is defined as a static method with a `this` modifier on the first parameter
- must be defined in a static class
- cannot access private instance variables in the extended class
- is called by - behind the scene - translating to a call of the static method

Exercise 5.5. *Extending struct Double*

At the accompanying page we have seen how class `Point` can be extended externally with the method `DistanceTo`. This is an *extension method*.

If you study the method `DistanceTo` you will see that we use the squareroot function `Math.Sqrt`, defined statically in class `Math`. And we could/should have used a similar `Square` function had it been available.

It is remarkable that C# 3.0 allows us to extend the structs behind the primitive types, such as `Double` and `Int32`.

Write a static class that extends struct `Double` with the two extension methods `Sqrt` and `Square`. Afterwards, rewrite the method `DistanceTo` such that it makes use of the new instance methods in struct `Double`.

20.11. Methods versus Properties versus Indexers

Lecture 5 - slide 28

Here at the end of the chapter about methods we will summarize some rule of thumbs for choosing between properties, indexers and methods in a class:

- Properties
  - For reading and extracting of individual instance/class variables
  - For writing and assigning individual instance/class variables
  - For other kinds of data access that does not involve time consuming computations
- Indexers
  - Like properties
  - Used when it is natural to access data by indexes - *array notation* - instead of simple names
  - Used as surface notation for associative arrays
- Methods
  - For all other operations that encapsulate calculations on the data of the class

20.12. References

[Bishop04] Judith Bishop and Nigel Horspool, *C# Concisely*. Pearson. Addison Wesley, 2004.

21. Overloaded Operators

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

21.1. Why operator overloading?

Lecture 6 - slide 2

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

Use of operators provides for substantial notational convenience in certain classes

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

21.2. Overloadable operators in C#

Lecture 6 - slide 3

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

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

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

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

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

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

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

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

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

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

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

21.3. An example of overloaded operators: Interval

Lecture 6 - slide 4

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

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

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

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

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

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

The private class `IntervalEnumerator` (shown only in the web version) and the method `GetEnumerator` make it possible to traverse an interval in a convenient way with use of **`foreach`**. Interval traversal is what makes intervals useful. This is illustrated in Program 21.4. We will, in great details, discuss `IntervalEnumerator` later in this material, see Section 31.6 - in particular Program 31.9.

```
using System;
using System.Collections;

public struct Interval{

  private readonly int from, to;

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

  public int From{
    get {return from;}
  }

  public int To{
    get {return to;}
  }

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

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

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

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

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

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

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

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

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

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

  private class IntervalEnumerator: IEnumerator{
    // Details not shown in this version
  }

  public IEnumerator GetEnumerator (){
    return new IntervalEnumerator(this);
  }

}
```

Program 21.3 *The struct Interval.*

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

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

```
using System;

public class app {

  public static void Main(){

    Interval iv1 = new Interval(17,14),
             iv2 = new Interval(2,5),
             iv3;

    foreach(int k in !(3 + iv1 - 2)){
      Console.Write("{0,4}", k);
    }
    Console.WriteLine();

    foreach(int k in !(3 + !iv2 * 2)){
      Console.Write("{0,4}", k);
    }
    Console.WriteLine();

    iv3 = !(3 + !iv2 * 3) >> 2 ;
    Console.WriteLine("First and last in iv3: {0}, {1}",
                       iv3[0], iv3[iv3.Length-1]);

    int[] arr = (int[])iv3;
    foreach(int j in arr){
      Console.Write("{0,4}", j);
    }

  }

}
```

Program 21.4 *A client program of struct Interval.*

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

Listing 21.5 *Output from the interval application.*

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

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

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

```
using System;

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


public class Card{

   private CardSuite suite;
   private CardValue value;

   // Some methods are not shown in this version

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

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

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

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

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

```
  }

  public static bool operator >(Card c1, Card c2){
    return !(c1 < c2) && !(c1 == c2);
  }
}
```

Program 21.6 *The class PlayingCard with relational operators.*

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

Exercise 6.1. *Interval indexer*

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

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

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

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

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

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

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

Exercise 6.2. *An interval overlap operation*

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

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

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

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

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

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

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

21.4. Some details of operator overloading

Lecture 6 - slide 5

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

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

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

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

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

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

22. Delegates

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

22.1. Delegates in C#

Lecture 6 - slide 7

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

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

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

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

```
using System;

public class Application {

  public delegate double NumericFunction(double d);

  public static void PrintTableOfFunction(NumericFunction f,
                                          string fname,
                                          double from, double to,
                                          double step){
    double d;

    for(d = from; d <= to; d += step){
     Console.WriteLine("{0,10}({1,-4:F3}) = {2}", fname, d, f(d));
    }

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

Program 22.1 *A Delegate of simple numeric functions.*

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

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

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

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

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

```
using System;

public class Application {

  public delegate double NumericFunction(double d);

  public static NumericFunction Compose
                   (NumericFunction f, NumericFunction g){
    return delegate(double d){return f(g(d));};
  }

  public static void PrintTableOfFunction
                   (NumericFunction f, string fname,
                    double from, double to, double step){
    double d;

    for(d = from; d <= to; d += step){
      Console.WriteLine("{0,35}({1,-4:F3}) = {2}", fname, d, f(d));
    }

    Console.WriteLine();
  }

  public static double Square(double d){
    return d*d;
  }

  public static double Cubic(double d){
    return d*d*d;
  }

  public static double Minus3(double d){
    return d-3;
  }

  public static void Main(){
    PrintTableOfFunction(Compose(Cubic, Minus3),
                         "Cubic of Minus3", 0.0, 5.0, 1.0);

    PrintTableOfFunction(
      Compose(Square, delegate(double d){
                       return d > 2 ? -d : 0;}),
      "Square of if d>2 then -d else 0", 0.0, 5.0, 1.0);
  }
}
```

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

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

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

Listing 22.4   *Output from the Compose delegate program.*

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

Delegates make it possible to approach the functional programming style

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

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

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

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

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

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

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

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

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

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

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

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

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

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

When we run the following program

```
using System;
public class Application {

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

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

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

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

we get this output

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

Explain!

22.2. Delegates that contain instance methods

Lecture 6 - slide 8

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

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

```
using System;

public delegate void Message(string txt);

public class Messenger{

  private string sender;
  private Message message;

  public Messenger(string sender){
   this.sender = sender;
   message = null;
  }

  public Messenger(string sender, Message aMessage){
   this.sender = sender;
   message = aMessage;
  }

  public void DoSend(){
   message("Message from " + sender);
  }
}
```

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

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

```
using System;

public class A{

  private int state;

  public A(int i){
    state = i;
  }

  public void MethodA(string s){
    Console.WriteLine("A: {0}, {1}", state, s);
  }
}
```

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

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

```
using System;

public class Application{

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

    Messenger m = new Messenger("CS at AAU", a2.MethodA);

    m.DoSend();
 }

}
```

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

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

Listing 22.8 *Output from Main of class Application.*

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

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

22.3. Multivalued delegates

Lecture 6 - slide 9

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

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

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

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

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

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

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

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

```
using System;

public class Application{

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

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

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

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

Program 22.10 *An Application class.*

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

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

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

Listing 22.11 *Output from Main of class Application.*

22.4. Lambda Expressions

Lecture 6 - slide 10

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

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

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

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

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

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

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

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

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

23. Events

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

23.1. Events

Lecture 6 - slide 12

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

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

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

Inversion of control

Don't call us - we call you

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

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

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

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

23.2. Events in C#

Lecture 6 - slide 13

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

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

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

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

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

23.3. Examples of events

Lecture 6 - slide 14

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

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

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

```
1  using System;
2  using System.Collections.Generic;
3
4  public delegate void Notifier(string message);
5
6  public class Die {
7
8    private int numberOfEyes;
9    private Random randomNumberSupplier;
10   private int maxNumberOfEyes;
11   private List<int> history;
12   public event Notifier twoSixesInARow;
13
14   public int NumberOfEyes{
15      get {return numberOfEyes;}
16   }
17
18   public Die (): this(6){}
19
20   public Die (int maxNumberOfEyes){
21     randomNumberSupplier = new Random(unchecked((int)DateTime.Now.Ticks));
22     this.maxNumberOfEyes = maxNumberOfEyes;
23     numberOfEyes = randomNumberSupplier.Next(1, maxNumberOfEyes + 1);
24     history = new List<int>();
25     history.Add(numberOfEyes);
26   }
27
28   public void Toss (){
29     numberOfEyes = randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
30     history.Add(numberOfEyes);
31     if (DoWeHaveTwoSixesInARow(history))
32        twoSixesInARow("Two sixes in a row");
33   }
34
35   private bool DoWeHaveTwoSixesInARow(List<int> history){
36     int histLength = history.Count;
37     return histLength >= 2 &&
38            history[histLength-1] == 6 &&
39            history[histLength-2] == 6;
40   }
41
42   public override String ToString(){
43     return String.Format("Die[{0}]: {1}", maxNumberOfEyes, NumberOfEyes);
44   }
45 }
```

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

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

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

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

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

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

```
using System;

class diceApp {

  public static void Main(){

    Die d1 = new Die();

    d1.twoSixesInARow +=
     delegate (string mes){
       Console.WriteLine(mes);
     };

    for(int i = 1; i < 100; i++){
      d1.Toss();
      Console.WriteLine("{0}: {1}", i, d1.NumberOfEyes);
    }

 }
}
```

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

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

```
1: 6
2: 4
...
32: 3
33: 6
Two sixes in a row
34: 6
Two sixes in a row
35: 6
...
66: 2
67: 6
Two sixes in a row
68: 6
69: 2
70: 4
...
97: 6
Two sixes in a row
98: 6
99: 3
```

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

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

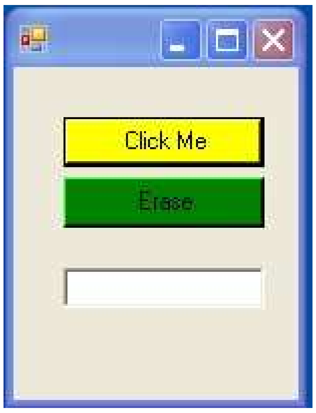

Figure 23.1 *A graphical user interface with two buttons and a textbox.*

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

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

public class Window: Form{

  private Button b1, b2;
  private TextBox tb;

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

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

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

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

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

}

class ButtonTest{

  public static void Main(){
    Window win = new Window();
    Application.Run(win);
  }

}
```

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

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

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

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

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

Exercise 6.5. *Additional Die events*

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

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

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

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

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

24. Patterns and Techniques

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

24.1. The observer design pattern

Lecture 6 - slide 17

The ***Observer*** is often used to ensure a *loose coupling* between an application and its user interface

In general, ***Observer*** can be used whenever a set of observer objects need to be informed about state changes in a subject object

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

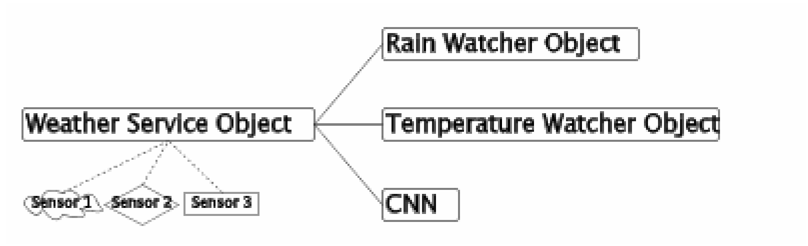

Figure 24.1 *The subject (weather service object) to the left and its three observers (weather watcher objects) to the right. The Weather Service Object get its information various sensors.*

The following questions are relevant:

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

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

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

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

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

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

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

 public class Subject {
   // Subject instance variables

   private ArrayList observers = new ArrayList();

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

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

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

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

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

Program 24.1 *Template of the Subject class.*

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

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

 public class Observer {

   private Subject mySubject;

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

   public void Update(){
```

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

Program 24.2   *A templates of the Observer class.*

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

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

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

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

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

24.2. Observer with Delegates and Events

Lecture 6 - slide 19

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

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

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

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

Program 24.4 *Template of the Subject class.*

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

 public class Observer {

   public Observer (){
     // ...
   }

   public void Update(SubjectState ss){
      //   if (the state ss is interesting){
```

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

Program 24.5 *Template of the Observer class.*

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

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

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

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

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.*

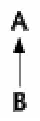

Figure 25.1 *The class B is a specialization of class A*

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*.

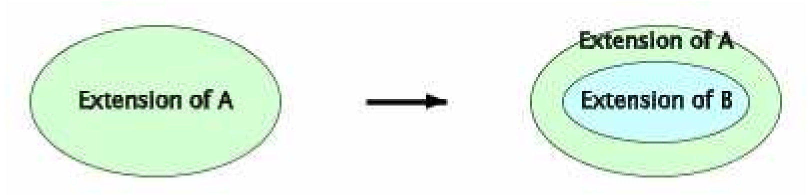

Figure 25.2 *The extension of a class A is narrowed when the class is specialized to B*

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`.

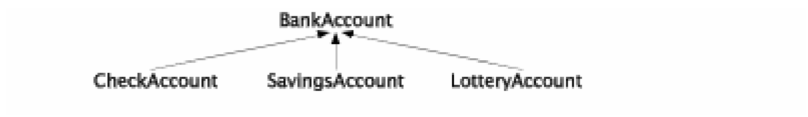

Figure 25.3 *A specialization hierarchy of bank accounts*

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.

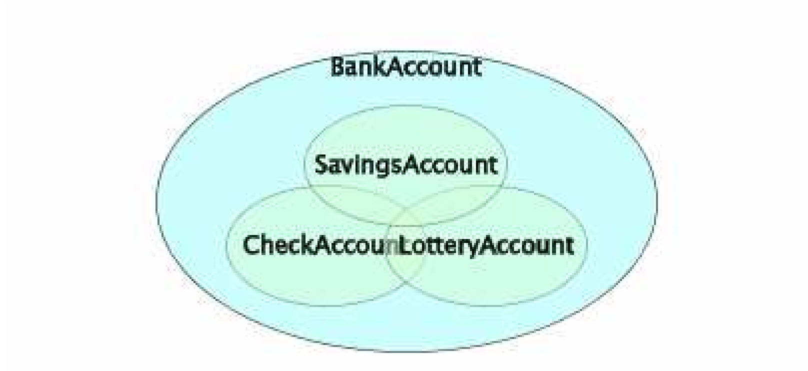

Figure 25.4 *Possible extensions of the bank account classes*

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.

```
using System;

public class BankAccount {

   protected double interestRate;
   protected string owner;
   protected decimal balance;

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

   public BankAccount(string o, double ir):
     this(o, 0.0M, ir) {
   }

   public virtual decimal Balance {
     get {return balance;}
   }

   public virtual void Withdraw (decimal amount) {
      balance -= amount;
   }

   public virtual void Deposit (decimal amount) {
      balance += amount;
   }

   public virtual void AddInterests() {
      balance += balance * (Decimal)interestRate;
   }

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

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***.

```
using System;

public class LotteryAccount: BankAccount {

   private static Lottery lottery  = Lottery.Instance(20);

   public LotteryAccount(string o, decimal b):
     base(o, b, 0.0) {
   }

   public override void AddInterests() {
      int luckyNumber = lottery.DrawLotteryNumber;
      balance = balance + lottery.AmountWon(luckyNumber);
   }

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

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.

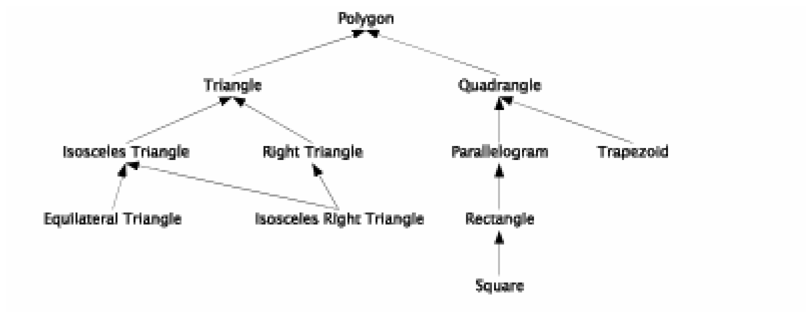

Figure 25.5 *A specialization hierarchy of polygons*

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.

```
using System;

public class Application{

  public static void Main(){
    Point2D p1 = new Point2D(1.1, 2.2),
            p2 = new Point2D(3.3, 4.4);

    Point3D q1 = new Point3D(1.1, 2.2, 3.3),
            q2 = new Point3D(4.4, 5.5, 6.6);

    p2.Move(1.0, 2.0);
    q2.Move(1.0, 2.0, 3.0);
    Console.WriteLine("{0} {1}", p1, p2);
    Console.WriteLine("{0} {1}", q1, q2);
  }

}
```

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

```
Point2D: (1,1, 2,2). Point2D: (4,3, 6,4).
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.

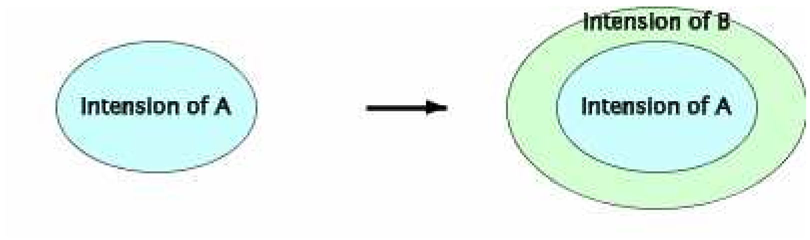

Figure 26.1 *The intension of a class A is blown up when the class is extended to B*

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.

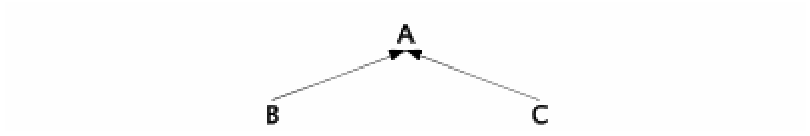

Figure 27.1 *Two classes B and C that inherit from class A*

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.

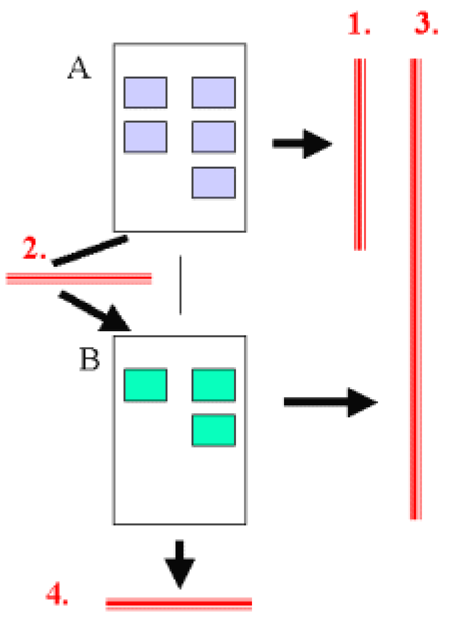

Figure 27.2 *Interfaces between A, B, their client classes, and their subclasses*

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.

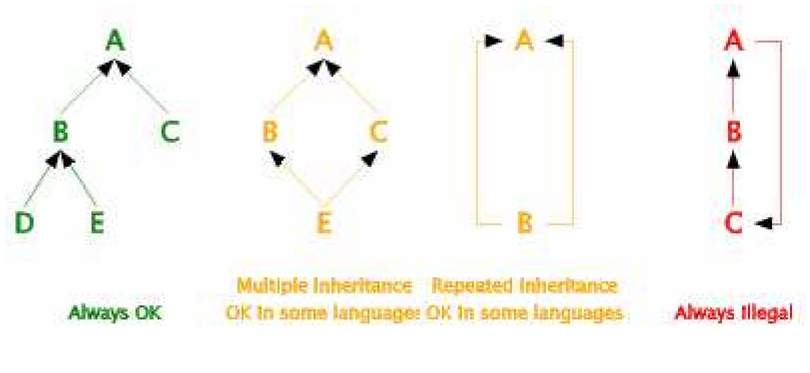

Figure 27.3 *Different graph structures among classes*

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).

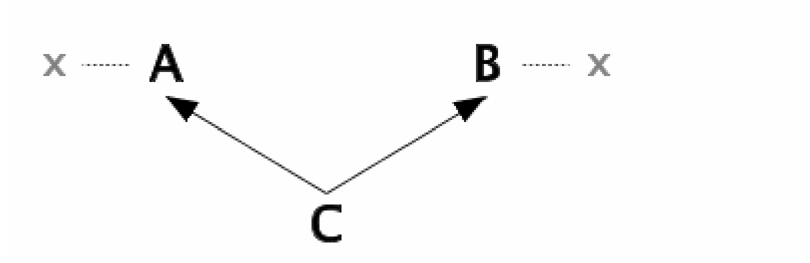

Figure 27.4 *Class B is a subclass of class A*

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.

```
class A {}

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.

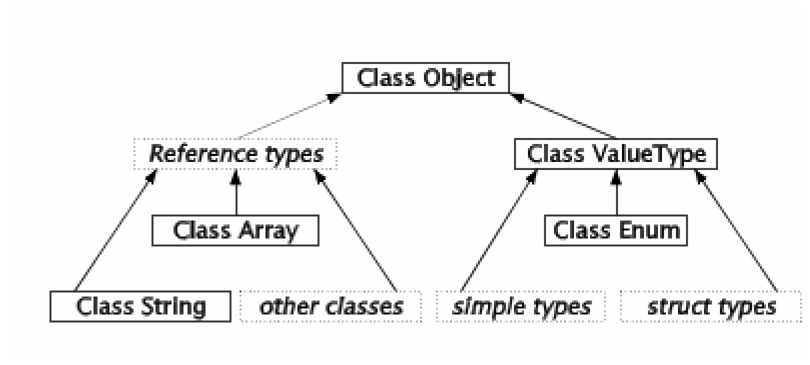

Figure 28.2 *The overall type hierarchy in C#*

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 
```

```
  public virtual decimal Balance {
    get {return balance;}
  }

  public virtual void Withdraw (decimal amount) {
     balance -= amount;
  }

  public virtual void Deposit (decimal amount) {
     balance += amount;
  }

  public virtual void AddInterests() {
     balance += balance * (Decimal)interestRate;
  }

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

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.

```
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 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.

```
using System;

public class C: B {
  private int varC1 = Init.InitMe(1, "varC1, initializer in class C"),
              varC2;

  public C (){
    varC2 = Init.InitMe(4, "VarC2, constructor body C");
  }
}
```

Program 28.6   *Initializers and constructors of class C.*

```
using System;

public class B: A {
  private int varB1 = Init.InitMe(1, "varB1, initializer in class B"),
              varB2;

  public B (){
    varB2 = Init.InitMe(4, "VarB2, constructor body B");
  }
}
```

Program 28.7 *Initializers and constructors of class B.*

```
using System;

public class A {
  private int varA1 = Init.InitMe(1, "varA1, initializer in class A"),
              varA2;

  public A (){
    varA2 = Init.InitMe(4, "VarA2, constructor body A");
  }
}
```

Program 28.8 *Initializers and constructors of class A.*

```
using System;

public class Init{

  public static int InitMe(int val, string who){
    Console.WriteLine(who);
    return val;
  }

}
```

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

```
using System;

class App{

  public static void Main(){
    C c = new C();
  }
}
```

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

```
varC1, initializer in class C
varB1, initializer in class B
varA1, initializer in class A
VarA2, constructor body A
VarB2, constructor body B
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.

```
using System;

public class BankAccount {

   protected double interestRate;
   protected string owner;
   protected decimal balance;

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

   public BankAccount(string o, double ir):
     this(o, 0.0M, ir) {
   }

   public virtual decimal Balance {
     get {return balance;}
   }

   public virtual void Withdraw (decimal amount) {
      balance -= amount;
   }

   public virtual void Deposit (decimal amount) {
      balance += amount;
   }

   public virtual void AddInterests() {
      balance += balance * (Decimal)interestRate;
   }

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

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.

```
using System;

public class CheckAccount: BankAccount {

   // Instance variables of BankAccount are inherited

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

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

   // Method Balance is inherited
   // Method Deposit is inherited
   // Method AddInterests is inherited

```

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

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

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.

```
class A {
  public void M(){}
}

class B: A{
  public void M(){}
}
```

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 **ba*i* 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.

```
N in A
N in B
N in B

P in A
P in A
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.

```
Per's check account holds 1030,000 kroner
Poul's savings account holds 930,000 kroner
Kurt's check account holds 1030,000 kroner
Bent's lottery account holds 1000,0 kroner
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`.

```
using System;
using System.Collections;

public class BankAccount {

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

   private static long nextAccountNumber = 0;
   private static ArrayList accounts = new ArrayList();

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

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

   public override bool Equals(Object obj){
     if (obj == null)
       return false;
     else if (this.GetType() != obj.GetType())
       return false;
     else if (ReferenceEquals(this, obj))
       return true;
     else if (this.accountNumber == ((BankAccount)obj).accountNumber)
       return true;
     else return false;
   }

   public override int GetHashCode(){
     return (int)accountNumber ^ (int)(accountNumber >> 32);
     // XOR of low orders and high orders bits of accountNumber
     // According to GetHashCode API recommendation.
```

```
  }

  /* Some methods are not included in this version */

}
```

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

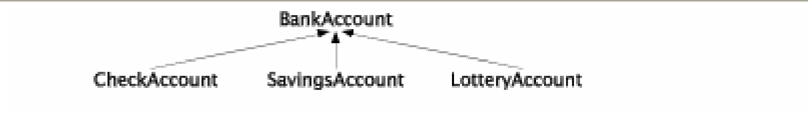

Figure 28.3 *A specialization hierarchy of bank accounts*

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.

29. Method Combination

In this section we will primarily study *method combination*. Secondarily we will touch on a more specialized, related problem called *parameter variance*.

29.1. Method Combination

Lecture 8 - slide 2

If two or more methods, of the same name, located different places in a class hierarchy, cooperate to solve some problem we talk about *method combination*.

A typical (and minimal) scene is outlined in Figure 29.1. Class B is a subclass of A, and in both classes there is a method named *Op*. Both *Op* methods have the same signature.

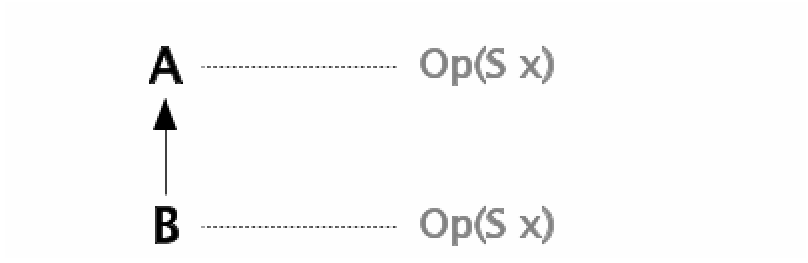

Figure 29.1 *Class B is a subclass of class A*

Overall, and in general, there are several ways for *Op* in class A and B to cooperate. We can, for instance, imagine that whenever a B-object receives an *Op* message, both operations are called automatically. We can also imagine that *Op* in class A is called explicitly by *Op* in class B, or the other way around.

Along the lines outlined above, we summarize two different method combination ideas. The first is known as *imperative method combination,* and the second is known as *declarative method combination*.

- Programmatic (imperative) control of the combination of *Op* methods
    - *Superclass controlled*: The *Op* method in class A controls the activation of *Op* in class B
    - *Subclass controlled*: The *Op* method in class B controls the activation of *Op* in class A
    - *Imperative method combination*
- An overall (declarative) pattern controls the mutual cooperation among *Op* methods
    - A.*Op* does not call B.*Op* - B.*Op* does not call A.*Op*.
    - A separate abstraction controls how *Op* methods in different classes are combined
    - *Declarative method combination*

Mainstream object-oriented programming languages, including C#, support imperative method combination. Most of them support the variant that we call subclass-controlled, imperative method combination.

Beta [Kristensen87] is an example of programming language with superclass-controlled, imperative method combination. CLOS [Steele90, Keene89] is one of the few examples of programming languages with declarative method combination. (The interested reader can consult Chapter 28 of [Steele90] to learn much more about declarative method combination in CLOS.)

C# supports subclass controlled, imperative method combination via use of the notation **`base.Op(...)`**

The notion **`base.Op(...)`** has been discussed in Section 28.7 and it has been illustrated in Program 26.2 (line 17), Program 28.13 (line 20), and Program 28.14 (line 20).

29.2. Parameter Variance

Lecture 8 - slide 3

We will continue the discussion of the scene outlined in Figure 29.1, now refined in Figure 29.2 shown below. The question is how the parameters of *Op* in class `A` and `B` vary in relation the variation of type `A` and type `B`.

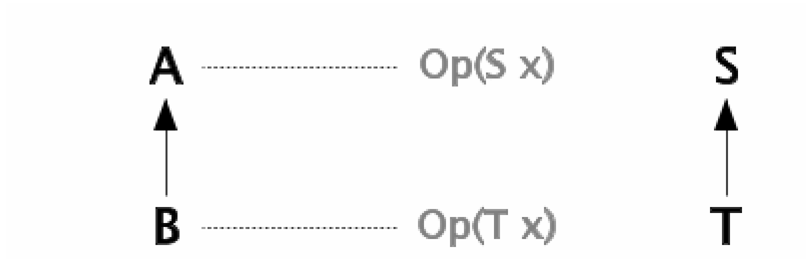

Figure 29.2 *Class B is a subclass of class A, and T is a subclass of S.*

In Program 29.1 we create an object of the specialized class `B` (in line 2), and we assign it to a variable of static type `A` (line 5) This is possible due to polymorphism. In line 6 we send the `Op` message to the `B` object. We assume that `Op` is virtual, and therefore we expect that `Op` in class `B` is called.

So far so good. The thing to notice is that `Op` takes a single parameter. If we pass an instance of class `S` to `B.Op` we may be in deep trouble. A problem occur if `B.Op` applies some operation from class `T` on the `S` object.

```
1    A aref;
2    B bref = new B();
3    S sref = new S();
4
5    aref = bref;     // aref is of static type A and dynamic type B
6    aref.Op(sref);   // B.Op is called with an S-object as parameter.
7                     // What if an operation from T is activated on the S-object?
```

Program 29.1 *An illustration of the problems with covariance.*

In Program 29.2 (only on web) in the web-edition we show a complete C# program which illustrates the problem.

The story told about the scene in Program 29.1 and Program 29.2 (only on web) turns out to be flawed in relation to C#! I could have told you the reason, but I will not do so right away. You should rather take a look at Exercise 8.1 and learn the lesson the hard way. (When access is granted to the exercise solutions, you will be able to get my explanation).

It turns out that parameter variance is not really a relevant topic in C#...

29.3. Covariance and Contravariance

Lecture 8 - slide 4

The situation encountered in Figure 29.2 of Section 29.2 is called *covariance*, because the types `S` and `T` (as occurring in the parameters of `Op` in class `A` and `B`) vary the same way as classes `A` and `B`. (The parameter type `T` of `Op` in class `B` is a subclass of the parameter type `S` of `Op` in class `A`; The class `B` is a subclass of class `A`; Therefore we say that `T` and `S` vary the same way as `A` and `B`.)

- **Covariance:** The parameters `S` and `T` vary the same way as `A` and `B`

As a contrast, the situation in Figure 29.3 below is called *contravariance*, because - in this variant of the scene - `S` and `T` vary in the opposite way as `A` and `B`. Please compare carefully Figure 29.2 with Figure 29.3.

- **Contravariance**: The parameters `S` and `T` vary the opposite way as `A` and `B`

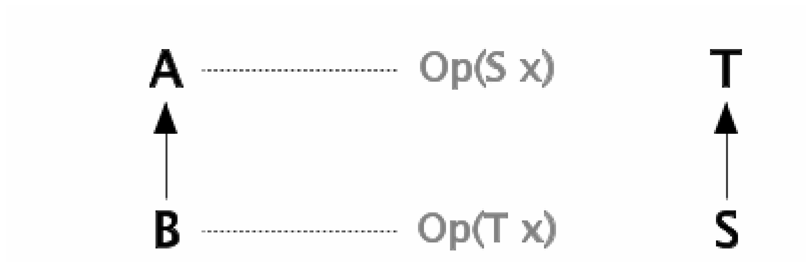

Figure 29.3 *Class B is a subclass of class A, and the parameter class S is a subclass of T.*

As we will see in Exercise 8.1 the distinction between covariance and contravariance is less relevant in C#. However, covariance and contravariance show up in other contexts of C#. See Section 42.6.

Exercise 8.1. *Parameter variance*

First, be sure you understand the co-variance problem stated above. Why is it problematic to execute `aref.Op(sref)`in the class Client?

The parameter variance problem, and the distinction between covariance and contravariance, is not really a topic in C#. The program with the classes A/B/S/T on the previous page compiles and runs without problems. Explain why!

29.4. References

[Keene89]	Sonya E. Keene, *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.
[Steele90]	Guy L. Steele, *Common Lisp, the language, 2nd Edition*. Digital Press, 1990.
[Kristensen87]	Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, "The BETA Programming Language". In *Research Directions in Object-Oriented Programming*, The MIT Press, 1987. Bruce Shriver and Peter Wegner (editors)

30. Abstract Classes - Sealed Classes

This chapter is about *abstract classes*. At the end of the chapter we also touch on *sealed classes*. Relative to our interests, sealed classes are less important than abstract classes.

30.1. Abstract Classes

Lecture 8 - slide 6

When we program in the object-oriented paradigm it is important to work out concepts as general as possible. Programming at a general level promotes reusability (see Section 2.4).

In object-oriented programming languages we organize classes in hierarchies. The classes closest to the root are the most general classes. Take, as an example, the bank account class hierarchy in Section 25.3, where the class `BankAccount` is more general than `CheckAccount`, `SavingsAccount`, etc. It is worth noticing, however, that we were able to fully implement all operations in the most general class, `BankAccount`. In the rest of this chapter we will study even more general classes, for which we cannot (or will not) implement all the operations. The non-implemented operations are stated as *declarations of intent* at the most general level. These declarations of intent should be realized in less general subclasses.

Abstract classes are used for concepts that we cannot or will not implement in full details

Here follows our definition of an abstract class and an abstract operation.

An *abstract class* is a class with one or more abstract operations

An *abstract operation* is specially marked operation with a name and with formal parameters, but without a body

An abstract class

- may announce a number of abstract operations, which must be supplied in subclasses
- cannot be instantiated
- is intended to be completed/finished in a subclass

We will sometimes use the term *concrete class* for a class which is not abstract.

You should be aware that the definition of an abstract class, as given below, is not 100% accurate in relation to C#. In C# a class can be abstract without announcing abstract operations. More about that in Section 30.2 below, where we discuss abstract classes in C#.

The fact that an abstract class cannot be instantiated is the most tangible, operational consequence of working with abstract classes. Many OOP programmers tend to think of the `abstract` modifier as a mark, to be associated with those classes, he or she does not wish to instantiate. Surely, this is a consequence, but it is not the essential idea behind abstract classes.

30.2. Abstract classes and abstract methods in C#

Lecture 8 - slide 7

We will first study an example of an abstract class. We pick an abstract stack. (This is indeed a very popular example class in many contexts. We have tried to avoid it, but here it fits nicely).

The abstract class `Stack`, shown in Program 30.1, is remarkable of two reasons:

1. There is no data representation in the class (no instance variables).
2. There is a fully implemented operation in the class, despite the fact that the class has no data for the operation to work on.

The **blue** parts of Program 30.1 are the abstract operations. These operations make up the classical stack operations `Push`, `Pop`, and `Top` together with `Full`, `Empty`, and `Size`. (Notice that `Top`, `Full`, `Empty` and `Size` are announced as properties, cf. Section 30.3). The abstract operations have signatures (method heads), but no body blocks. In a real-life version of the program we would certainly have supplied documentation comments with some additional explanations of the roles of the abstract operations in the class.

The **purple** part represents a fully implemented, "normal" method, called `ToggleTop`. This method swaps the order of the two top-most elements of the stack (if available). Notice that `ToggleTop` can be implemented solely in terms of the `Push`, `Pop`, `Top` and `Size`. In other words, it is not necessary for the implementation of `ToggleTop` to know details of the concrete data representation of stacks.

```
using System;

public abstract class Stack{

  abstract public void Push(Object el);

  abstract public void Pop();

  abstract public Object Top{
    get;}

  abstract public bool Full{
    get;}

  abstract public bool Empty{
    get;}

  abstract public int Size{
    get;}

  public void ToggleTop(){
    if (Size >= 2){
      Object topEl1 = Top;  Pop();
      Object topEl2 = Top;  Pop();
      Push(topEl1); Push(topEl2);
    }
  }

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

Program 30.1 *An abstract class Stack - without data representation - with a non-abstract ToggleTop method.*

In Program 30.1 the method `ToString` is also an example of a fully implemented method, which relies on an abstract method, namely `Size`.

It is left as an exercise to implement a non-abstract subclass of the abstract stack, see Exercise 8.3.

Let us state some more detailed - a perhaps slightly surprising - observations about abstract classes and abstract operations. Each of them will be discussed below.

- Abstract classes
  - can be derived from a non-abstract class
  - do not need not to have abstract members
  - can have constructors
- Abstract methods
  - are implicitly virtual

In relative rare situations an abstract class can inherit from a non-abstract class. Notice, however, that even abstract classes inherit (at least implicitly) from class `Object`, which is a non-abstract class in C#. (In principle, it would make good sense for the designers of C# to implement class `Object` as abstract class. But they did not! We only rarely make instances of class `Object`).

The next observation is about fully implemented classes, which we mark as being abstract. As discussed above, the purpose of this marking is to prevent instantiation of the class.

You may ask if it makes sense to have constructors in a class which never is instantiated. The answer is yes, because the data encapsulated in an abstract class `A` should be initialized when a concrete subclass of `A` is instantiated. Due to the rules of constructor cooperation, see Section 28.4 and Section 28.5, a constructor of class `A` will be activated. If no constructor is present in `A`, this falls back on the parameter-less default constructor.

Finally, we observe that the abstract methods are implicitly virtual. This is natural, because such a method has to be (re)defined in a subclass. In C# it is not allowed explicitly to write "`virtual abstract`" in front of an abstract method. Let us also observe, that an abstract method *M* cannot be private. This is because *M* need to be visible in the classes that override *M*.

Exercise 8.2. *Course and Project classes*

In the earlier exercise about courses and projects (found in the lecture about classes) we programmed the classes `BooleanCourse`, `GradedCourse`, and `Project`. Revise and reorganize your solution (or the model solution) such that `BooleanCourse` and `GradedCourse` have a common abstract superclass called `Course`.

Be sure to implement the method `Passed` as an abstract method in class `Course`.

In the `Main` method (of the client class of `Course` and `Project`) you should demonstrate that both boolean courses and graded courses can be referred to by variables of static type `Course`.

Exercise 8.3. *A specialization of Stack*

On the slide to which this exercise belongs, we have shown an abstract class `Stack`.

It is noteworthy that the abstract `Stack` is programmed without any instance variables (that is, without any data representation of the stack). Notice also that we have been able to program a single non-abstract method `ToggleTop`, which uses the abstract methods `Top`, `Pop`, and `Push`.

Make a non-abstract specialization of `Stack`, and decide on a reasonable data representation of the stack.

In this exercise it is OK to ignore exception/error handling. You can, for instance, assume that the capacity of the stack is unlimited; That popping an empty stack an empty stack does nothing; And that the top of an empty stack returns the string "Not Possible". In a later lecture we will revisit this exercise in order to introduce exception handling. Exception handling is relevant when we work on full or empty stacks.

Write a client of your stack class, and demonstrate the use of the inherited method `ToggleTop`. If you want, you can also adapt my stack client class which is easily available to you in the web-edition of this material.

30.3. Abstract Properties

Lecture 8 - slide 8

Properties were introduced in Chapter 18. Recall that properties allow us to get and set data of a class through getter and setter abstractions. From an application point of view, properties are used in the same way as variables - both on the left and right hand sides of assignments. Underneath, a property is realized as two methods - one "getter" and one "setter".

Properties can be abstract in the same way as methods. It means that we can announce a number of properties which must be fully defined in subclasses. We will in Program 30.2 study an example of abstract properties, namely in a `Point` class called `AbstractPoint`, which can be accessed both in a rectangular (*x*, *y*) and a polar (*r*, *a*) way. *r* and *a* means radius and angle respectively. There is no data (variables) in class `AbstractPoint`. We announce `X`, `Y`, `R` and `A` as abstract properties. These are emphasized using **purple** color. All of these are announced as both getters and setters. Notice the `get; set;` syntax. We could alternatively announce these as only getters, or as only setters. We notice that the syntax of abstract properties is similar to the syntax used for automatic properties, see Section 18.3.

Following the abstract properties comes three noteworthy methods `Move`, `Rotate` and `ToString`. They are shown in **blue**. They all use make heavy use the abstract properties. The assignment `X += dx` in Move, for instance, expands to `X = X + dx`. It first uses the getter of the `X` property on the right hand side of the assignment. Next, it uses the `X` setter on the left hand side. In Program 30.2 we only know that the `X` getter and the `X` setter exist. The actual implementation details will be found in a subclass.

In the web-edition of this material, we show a version of class `AbstractPoint` with four additional protected, static methods which are useful for the implementation of the subclasses.

```
using System;

abstract public class AbstractPoint {

  public enum PointRepresentation {Polar, Rectangular}

  // We have not yet decided on the data representation of Point

  public abstract double X {
    get ;
    set ;
  }

  public abstract double Y {
    get ;
    set ;
  }

  public abstract double R {
    get ;
    set ;
  }

  public abstract double A {
    get ;
    set ;
  }

  public void Move(double dx, double dy){
    X += dx;  Y += dy;
  }

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

  public override string ToString(){
    return  "(" + X + ", " + Y + ")" + " " +  "[r:" + R + ", a:" + A + "]  ";
  }

}
```

Program 30.2 *The abstract class Point with four abstract properties.*

In Program 30.3 we see a subclass of `AbstractPoint`. It is called `Point`. It happens to represent points the polar way. But this is an internal (private) detail of class `Point`.

Class `Point` is a non-abstract class, and therefore we program a constructor, which is emphasized in **black**. The constructor is a little unconventional, because the first parameter allows us to specify if parameter two and three means `x`, `y` or `radius`, `angle`. It is desirable if this could be done more elegantly. (It can! Use of static factory methods, see Section 16.4, is better). Notice that `PointRepresentation` is an enumeration type defined in line 5 of Program 30.2.

Emphasized in **purple** we show the actual implementation of the `X` and `Y` properties. Let us look at `X`. The getter of `X` is called whenever `X` is used as a right-hand side value. It calculates the x-coordinate of a point from the radius and the angle. The setter of `X` is called when `X` is used in left-hand side context, such as `X = `*e*. The value of expression *e* is bound to the pseudo variable **`value`**. The setter calculates new radius and angle values which are assigned to the instance variables of class `Point`.

Emphasized in **blue** we show the implementation of the `R` and `A` properties. These are trivial compared to the `X` and `Y` properties, because we happen to represent points in the polar way.

```
using System;

public class Point: AbstractPoint {

  // Polar representation of points:
  private double radius, angle;            // radius, angle

  // Point constructor:
  public Point(PointRepresentation pr, double n1, double n2){
    if (pr == PointRepresentation.Polar){
      radius = n1; angle = n2;
    }
    else if (pr == PointRepresentation.Rectangular){
      radius = RadiusGivenXy(n1, n2);
      angle  = AngleGivenXy(n1, n2);
    } else {
      throw new Exception("Should not happen");
    }
  }

  public override double X {
    get {
      return XGivenRadiusAngle(radius, angle);}
    set {
      double yBefore = YGivenRadiusAngle(radius, angle);
      angle = AngleGivenXy(value, yBefore);
      radius = RadiusGivenXy(value, yBefore);
    }
  }

  public override double Y {
    get {
      return YGivenRadiusAngle(radius, angle);}
    set {
      double xBefore = XGivenRadiusAngle(radius, angle);
      angle = AngleGivenXy(xBefore, value);
      radius = RadiusGivenXy(xBefore, value);
    }
  }

  public override double R {
    get {
     return radius;}
    set {
     radius = value;}
  }

  public override double A {
    get {
     return angle;}
    set {
     angle = value;}
  }

}
```

Program 30.3 *A non-abstract specialization of class Point (with private polar representation).*

In the web-edition we show a client of `AbstractPoint` and `Point`, which is similar to Program 11.3 from Section 11.6. It shows how to manipulate instances of class `Point` via its abstract interface.

Let us summarize what we have learned from the examples in Program 30.2, Program 30.3, and Program 30.4 (only on web). First and foremost, we have seen an abstract class in which we are able to implement useful functionality (`Move`, `Rotate`, and `ToString`) *at a high level of abstraction*. The implementation details in the mentioned methods rely on abstract properties, which are implemented in subclasses. We have also seen a sample subclass that implements the four abstract properties.

30.4. Sealed Classes and Sealed Methods

Lecture 8 - slide 9

We will now briefly, as the very last part of this chapter, describe sealed classes and sealed methods.

A sealed class C prevents the use of C as base class of other classes

- Sealed classes
    - Cannot be inherited by other classes
- Sealed methods
    - Cannot be redefined and overridden in a subclass
    - The modifier **`sealed`** must be used together with **`override`**

Sealed classes are related to static classes, see Section 11.12, in the sense that none of them can be subclassed. However, static classes are more restrictive because a static class cannot have instance members, a static class cannot be used as a type, and a static class cannot be instantiated. Sealed classes and methods correspond to final classes and final methods in Java.

In some sense, abstract and sealed classes represent opposite concepts. At least this holds in the following sense: A sealed class cannot be subclassed; An abstract class must be subclassed in order to be useful.

If a class is abstract it does not make sense that it is sealed. And the other way around, if a class is sealed it does not make sense that it, in addition, is abstract. Notice that it does not make sense either to have virtual methods in a sealed class.

A sealed class is not required to have sealed methods. Moreover, a class with a sealed method does not itself need to be sealed.

Finally, notice, that in C# a method cannot be sealed without also being overridden. Thus, the **`sealed`** modifier always occurs as an "extra modifier" of **`override`**. The intention of sealed methods is to prevent further overriding of virtual methods.

31. Interfaces

Interfaces form a natural continuation of abstract classes, as discussed in Chapter 30. In this chapter we will first introduce the interface concept. Then follows an example, which illustrates the power of interfaces. Finally, we review the use of interfaces in the C# libraries.

31.1. Interfaces

Lecture 8 - slide 11

An interface announces a number of operations in terms of their signatures (names and parameters). An interface does not implement any of the announced operations. An interface only declares an intent, which eventually will be realized by a class or a struct.

A class or struct can implement an arbitrary number of interfaces. Inheritance of multiple classes may be problematic in case the same variable or (fully defined) operation is inherited from several superclasses, see Section 27.5. Inheritance of the same intent from multiple interfaces is less problematic. In a nutshell, this explains one of the reasons behind having interfaces in C# and Java, instead of having multiple class-inheritance, like in for instance C++ and Eiffel.

An interface can inherit an arbitrary number of other interfaces. This makes it convenient to organize a small set of inter-dependent operations in a single interfaces, which then can be combined (per inheritance) with several other interfaces, classes or structs.

An interface corresponds to a class where all operations are abstract, and where no variables are declared. In Section 30.1 we argued that abstract classes are useful as general, high-level program contributions. This is therefore also the case for interfaces.

An *interface* describes signatures of operations, but it does not implement any of them

Here follows the most important characteristics of interfaces:

- Classes and structs can implement one or more interfaces
- An interface can be used as a type, just like classes
    - Variables and parameters can be declared of interface types
- Interfaces can be organized in multiple inheritance hierarchies

Let us dwell on the observation that an interface serves as a type. We already know that classes and structs can be used as types. It means that we can have variables and parameters declared as class or struct types. The observation from above states that interfaces can be used the same way. Thus, it is possible to declare variables and parameters of an interface type. But wait a moment! It is not possible to instantiate an interface. So which values can be assigned to variables of an interface type? The answer is that objects or values of class of struct types, which implement the interface, can be assigned to variables of the interface type. This gives a considerable flexibility in the type system, because arbitrary types in this way can be made compatible, by letting them implement the same interface(s). We will see an example of that in Section 31.3.

Exercise 8.4. *The interface ITaxable*

For the purpose of this exercise you are given a couple of very simple classes called `Bus` and `House`. Class `Bus` specializes the class `Vehicle`. Class `House` specializes the class `FixedProperty`. The mentioned classes can easily be accessed from the web-edition of the material..

First in this exercise, program an interface `ITaxable` with a parameterless operation `TaxValue`. The operation should return a decimal number.

Next, program variations of class `House` and class `Bus` which implement the interface `ITaxable`. Feel free to invent the concrete taxation of houses and busses. Notice that both class `House` and `Bus` have a superclass, namely `FixedProperty` and `Vehicle`, respectively. Therefore it is essential that taxation is introduced via an interface.

Demonstrate that taxable house objects and taxable bus objects can be used together as objects of type `ITaxable`.

31.2. Interfaces in C#

Lecture 8 - slide 12

Let us now be more specific about interfaces in C#. The operations, described in a C# interface, can be methods, properties, indexers, or events.

Both classes, structs and interfaces can implement one or more interfaces

Interfaces can contain signatures of methods, properties, indexers, and events

The syntax involved in definition of C# interfaces is summarized in Syntax 31.1. The first few lines describe the structure of an interface as such. The remaining part of Syntax 31.1 outlines the descriptions of interface methods, properties, indexers and events respectively.

```
modifiers interface interface-name : base-interfaces {
  method-descriptions
  property-descriptions
  indexer-descriptions
  event-descriptions
}

return-type method-name(formal-parameter-list);

return-type property-name{
  get;
  set;
}

return-type this[formal-parameter-list]{
  get;
  set;
}
```

```
event delegate-type event-name;
```

Syntax 31.1 *The syntax of a C# interface, together with the syntaxes of method, property, indexer, and event descriptions in an interface*

31.3. Examples of Interfaces

Lecture 8 - slide 13

Earlier in this material we have programmed dice and playing cards, see Program 10.1 and Program 12.7. Do the concepts behind these classes have something in common? Well - they are both used in a wide variety of games. This observation causes us to define an interface, `IGameObject`, which is intended to capture some common properties of dice, playing cards, and other similar types. Both class `Die` and class `Card` should implement the interface `IGameObject`.

As a matter of C# coding style, all interfaces start with a capital 'I' letter. This convention makes it obvious if a type is defined by an interface. This naming convention is convenient in C#, because classes and interface occur together in the inheritance clause of a class. (Both the superclass and the interfaces occur after a colon, the class first, cf. Syntax 28.1). In this respect, C# is different from Java. In Java, interfaces and classes are marked with the keywords **`extends`** and **`implements`** respectively in the inheritance clause of a class.

Two or more unrelated classes can be used together if they implement the same interface

```
public enum GameObjectMedium {Paper, Plastic, Electronic}

public interface IGameObject{

  int GameValue{
    get;
  }

  GameObjectMedium Medium{
    get;
  }
}
```

Program 31.1 *The interface IGameObject.*

The `IGameObject` interface in Program 31.1 prescribes two named properties: `GameValue` and `Medium`. Thus, classes that implement the `IGameObject` must define these two properties. Notice, however, that no semantic constraints on `GameValue` or `Medium` are supplied. (It means that no *meaning* is prescribed). Thus, classes that implement the interface `IGameObject` are, in principle, free to supply arbitrary bodies of `GameValue` and `Medium`. This can be seen as a weakness. In Chapter 50 we will see how to remedy this by specifying the semantics of operations in terms of preconditions and postconditions.

Notice also that there are no visibility modifiers of the operations `GameValue` and `Medium` in the interface shown above. All operations are implicitly public.

Below, in Program 31.2, we show a version of class `Die`, which implements the interface `IGameObject`. In line 3 it is stated that class `Die` implements the interface. The actual implementations of the two operations are shown in the bottom part of Program 31.2 (from line 33 to 44). Most interesting, the `GameValue` of a die is the current number of eyes.

```
using System;

public class Die: IGameObject {
  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();
  }

  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("Die[{0}]: {1}", maxNumberOfEyes, numberOfEyes);
  }

  public int GameValue{
    get{
      return numberOfEyes;
    }
  }

  public GameObjectMedium Medium{
    get{
      return
       GameObjectMedium.Plastic;
    }
  }

}
```

Program 31.2 *The class Die which implements IGameObject.*

In Program 31.3 we show a version of class `Card`, which implements our interface. The `GameValue` of a card is, quite naturally, the card value.

```
using System;

public class Card: IGameObject{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  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);
  }

  public int GameValue{
    get { return (int)(this.value); }
  }

  public GameObjectMedium Medium{
    get{
      return GameObjectMedium.Paper;
    }
  }
}
```

Program 31.3 *The class Card which implements IGameObject.*

Below, in Program 31.4 we have written a program that works on game objects of type `IGameObject`. In order to be concrete - and somewhat realistic - we make an `IGameObject` array with three die objecs and three card objects. In the bottom part of the program we exercise the common operations of dice and playing cards, as prescribed by the interface `IGameObject`. The output of the program is shown in Listing 31.5.

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

class Client{

  public static void Main(){

    Die d1 = new Die(),
        d2 = new Die(10),
        d3 = new Die(18);

    Card c1 =  new Card(Card.CardSuite.spades, Card.CardValue.queen),
         c2 =  new Card(Card.CardSuite.clubs, Card.CardValue.four),
         c3 =  new Card(Card.CardSuite.diamonds, Card.CardValue.ace);

    IGameObject[] gameObjects = {d1, d2, d3, c1, c2, c3};

    foreach(IGameObject gao in gameObjects){
      Console.WriteLine("{0}: {1} {2}",
                        gao, gao.GameValue, gao.Medium);
    }
  }
}
```

Program 31.4 *A sample Client program of Die and Card.*

```
Die[6]: 5: 5 Plastic
```

```
Die[10]: 9: 9 Plastic
Die[18]: 15: 15 Plastic
Suite:spades, Value:queen: 12 Paper
Suite:clubs, Value:four: 4 Paper
Suite:diamonds, Value:ace: 14 Paper
```

Listing 31.5 *Output from the sample Client program of Die and Card.*

Above, both `Die` (see Program 31.2) and `Card` (see Program 31.3) are classes. We have in Exercise 4.2 noticed that it would be natural to implement the type `Card` as a struct, because a playing card - in contrast to a die - is immutable. The client class shown in Program 31.4 will survive if we program `Card` as a struct, and it will produce the same output as shown in Listing 31.5. Recall in this context that interfaces in C# are reference types, see Section 13.3. When a variable of static type `IGameObject` is assigned to a value of struct type `Card`, the card value is boxed. Boxing is described in Section 14.8.

In the example above, where both the types `Die` and `Card` are implemented as classes, `IGameObject` could as well have been implemented as an abstract superclass. This is the theme in Exercise 8.5.

Exercise 8.5. *An abstract GameObject class*

On the slide, to which this exercise belongs, we have written an interface `IGameObject` which is implemented by both class `Die` and class `Card`.

Restructure this program such that class `Die` and class `Card` both inherit an abstract class `GameObject`. You should write the class `GameObject`.

The client program should survive this restructuring. (You may, however, need to change the name of the type `IGameObject` to `GameObject`). Compile and run the given client program with your classes.

31.4. Interfaces from the C# Libraries

Lecture 8 - slide 14

The C# library contains a number of important interfaces which are used frequently in many C# programs

In this section we will discuss some important interfaces from the C# libraries. First, we give an itemized overview, and in the sections following this one more details will be provided.

- `IComparable`
    - An interface that prescribes a `CompareTo` method
    - Used to support general sorting and searching methods
- `IEnumerable`
    - An interface that prescribes a method for accessing an enumerator
- `IEnumerator`
    - An interface that prescribes methods for traversal of data collections
    - Supports the underlying machinery of the **`foreach`** control structure
- `IDisposable`

    - An interface that prescribes a `Dispose` method
    - Used for deletion of resources that cannot be deleted by the garbage collector
    - Supports the C# **`using`** control structure
- `ICloneable`
    - An interface that prescribes a `Clone` method
- `IFormattable`
    - An interface that prescribes an extended `ToString` method

`IComparable` is touched on in Section 31.5, primarily via an exercise. In Section 31.6 we focus on the interfaces `IEnumerable` and `IEnumerator` and their roles in the realization of **`foreach`** loops. Type parameterized versions of these interfaces are discussed in Section 45.2. The interfaces `IDisposable` is discussed in the context of IO in Section 37.5. `ICloneable` is discussed in a later chapter, see Section 32.7.

All the interfaces mentioned above can be thought of as *flavors* that can be added to many different classes.

31.5. Sample use of IComparable

Lecture 8 - slide 15

Object of classes that implement **`IComparable`** can be sorted by a method such as **`Array.Sort`**

In many contexts it is important to be able to state that two objects or values, say *x* and *y* of a particular type *T*, can be compared to each other. Thus, we may be curious to know if $x < y$, $y < x$, or if $x = y$. But what does $x < y$, $y > x$, and $x = y$ mean if, for instance type *T* is `BankAccount` or a `Die`?

The way we approach this problem is to arrange that the type *T* (a class or a struct) implements the interface `IComparable`. In that way, the implementation of *T* must include the method `CompareTo`, which can be used in the following way:

```
x.CompareTo(y)
```

In the tradition of, for instance, the string comparison function `strcmp` in the C standard library `string.h` the expression *x*`.CompareTo(`*y*`)` returns a negative integer result if *x* is considered less than *y*, a positive integer if *x* is considered greater than *y*, and integer zero if *x* and *y* are considered to be equal.

The interface `IComparable` is reproduced in Program 31.6. This shows you how simple it is. Don't use this or a similar definition. Use the interface `IComparable` as predefined in the `System` namespace.

```
using System;

public interface IComparable{
  int CompareTo(Object other);
}
```

Program 31.6 *A reproduction of the interface IComparable.*

The parameter of `CompareTo` is of type `Object`. This is irritating because we will almost certainly want the parameter to be of the same type as the class, which implements `Icomparable`. When you solve Exercise 8.6 you will experience this.

There is actually two versions of the interface `IComparable` in the C# libraries. The one similar to Program 31.6 and a type parameterized version, which constrains the parameter of the `CompareTo` method to a given type `T`. We have more say about these two interfaces in Section 42.8.

It is also worthwhile to point out the interface `IEquatable`, which simply prescribes an `Equals` method. The interface `IEqualityComparer` is a cousin interface which in addition to `Equals` also prescribes `GetHashCode`. In some sense `IEquatable` and `IEqualityComparer` are more fundamental than `IComparable`. It turns out that `IEquatable` only exists as a type parameterized (generic) interface.

Exercise 8.6. *Comparable Dice*

In this exercise we will arrange that two dice can be compared to each other. The result of `die1.CompareTo(die2)` is an integer. If the integer is negative, `die1` is considered less than `die2`; If zero, `die1` is considered equal to `die2`; And if positive, `die1` is considered greater than `die2`. When two dice can be compared to each other, it is possible sort an array of dice with the standard `Sort` method in C#.

Program a version of class `Die` which implements the interface `System.IComparable`.

Consult the documentation of the (overloaded) static method `System.Array.Sort` and locate the `Sort` method which relies on `IComparable` elements.

Make an array of dice and sort them by use of the `Sort` method.

31.6. Sample use of IEnumerator and IEnumerable

Lecture 8 - slide 16

In this section we will study the interfaces called `IEnumerator` and `IEnumerable`. The interface `IEnumerator` is central to the design pattern called ***Iterator***, which we will discuss in the context of collections, see Section 48.1. As already mentioned above, the interface `IEnumerator` also prescribes the operations behind the **`foreach`** control structure.

```
using System;

public interface IEnumerator{
  Object Current{
    get;
  }

  bool MoveNext();

  void Reset();
}
```

Program 31.7 *A reproduction of the interface IEnumerator.*

We have reproduced IEnumerator from the System.Collections namespace in Program 31.7. The operations Current, MoveNext, and Reset are used to traverse a collection of data. Hidden behind the interface should be some simple bookkeeping which allows us to keep track of the current element, and which element is next. You can think of this as a *cursor*, which step by step is moved through the collection. The property Current returns the element pointed out by the cursor. The method MoveNext advances the cursor, and it returns true if it has been possible to move the cursor. The method Reset moves the cursor to the first element, and it resets the bookkeeping variables.

You are not allowed to modify the collection while it is traversed via a C# enumerator. Notice, in particular, that you are not allowed to delete the element obtained by Current during a traversal. In that respect, C# enumerators are more limited than the Iterator counterpart in Java which allows for exactly one deletion for each movement in the collection. It can also be argued that the IEnumerator interface is too narrow. It would be nice to have a boolean HasNext property. It could also be worthwhile to have an extended enumerator with a MovePrevious operation.

Like it was the case for the interface Comparable, as discussed in Section 31.5, there is also a type parameterized version of IEnumerator. See Section 45.2 for additional details.

```
using System.Collections;

public interface IEnumerable{
  IEnumerator GetEnumerator();
}
```

Program 31.8 *A reproduction of the interface IEnumerable.*

The IEnumerable interface, as reproduced in Program 31.8, only prescribes a single method called GetEnumerator. This method is intended to return an object (value), the class (struct) of which implements the IEnumerator interface. Thus, if a type implements the IEnumerable interface, it can deliver an iterator/enumerator object via use of the operation GetEnumerator.

As mentioned above, the **foreach** control structure is implemented by means of enumerators. The **foreach** form

```
  foreach(ElementType e in collection) statement
```

is roughly equivalent with

```
  IEnumerator en = collection.GetEnumerator();
  while (en.MoveNext()){
    ElementType e = (ElementType) en.Current();
    statement;
  }
```

The type of the collection is assumed to implement the interface **IEnumerable**. Additional fine details should be taken into consideration. Please consult section 15.8.4 of the C# Language Specification [ECMA-334] or [Hejlsberg06] for the full story.

We will now present a realistic example that uses IEnumerator and IEnumerable. We return to the Interval type, which we first met when we discussed overloaded operators in Section 21.3. The original Interval struct appeared in Program 21.3. Recall that an interval, such as [5 - 10] is different from [10 -5]. The former represents the sequence 5, 6, 7, 8, 9, 10 while the latter represents 10, 9, 8, 7, 6, 5. In the version we show in Program 31.9 we have elided the operators from Program 21.3.

The enumerator functionality is programmed in a private, local class called `IntervalEnumerator`, starting at line 39. This class implements the interface `IEnumerator`. The class `IntervalEnumerator` has a reference to the surrounding interval. (The reference to the surrounding object is provided via the constructor in line 44 and 68). It also has the instance variable `idx`, which represents of the cursor. Per convention, the value -1 represents an interval which has been reset. The property `Current` is now able to calculate and return a value from the interval. Notice that we have to distinguish between rising and falling intervals in the conditional expression in line 50-52. Both `MoveNext` and `Reset` are easy to understand if you have followed the details until this point.

The method `GetEnumerator` (line 67-69), which is prescribed by the interface, `IEnumerable` (see line 4), just returns an instance of the private class `IntervalEnumerator` discussed above. Notice that we in line 68 pass `this` (the current instance of the `Interval`) to the `IntervalEnumerator` object.

We show how to make simple traversals of intervals in Program 31.10.

```
1  using System;
2  using System.Collections;
3
4  public struct Interval: IEnumerable{
5
6    private readonly int from, to;
7
8    public Interval(int from, int to){
9      this.from = from;
10     this.to = to;
11   }
12
13   public int From{
14     get {return from;}
15   }
16
17   public int To{
18     get {return to;}
19   }
20
21   public int Length{
22     get {return Math.Abs(to - from) + 1;}
23   }
24
25   public int this[int i]{
26     get {if (from <= to){
27            if (i >= 0 && i <= Math.Abs(from-to))
28                return from + i;
29            else throw new Exception("Error"); }
30          else if (from > to){
31            if (i >= 0 && i <= Math.Abs(from-to))
32                return from - i;
33            else throw new Exception("Error"); }
34          else throw new Exception("Should not happen"); }
35   }
36
37   // Overloaded operators have been hidden in this version
38
39   private class IntervalEnumerator: IEnumerator{
40
41     private readonly Interval interval;
42     private int idx;
43
44     public IntervalEnumerator (Interval i){
45       this.interval = i;
46       idx = -1;   // position enumerator outside range
47     }
48 
49     public Object Current{
50          get {return (interval.From < interval.To) ?
51                        interval.From + idx :
52                        interval.From - idx;}
53     }
54 
55     public bool MoveNext (){
56       if ( idx < Math.Abs(interval.To - interval.From))
57          {idx++; return true;}
58       else
59          {return false;}
60     }
61 
62     public void Reset(){
63       idx = -1;
64     }
65   }
66 
67   public IEnumerator GetEnumerator (){
68     return new IntervalEnumerator(this);
69   }
70 
71 }
```

Program 31.9 *IEnumerator in the type Interval.*

While we are here, we will discuss the nested, local class `IntervalEnumerator` of class `Interval` a little more careful. Why is it necessary to pass a reference to the enclosing `Interval` in line 68? Or, in other words, why can't we access the `from` and `to` `Interval` instance variables in line 6 from the nested class? The reason is that an `IntervalEnumerator` object is not a 'subobject' of an `Interval` object. An `IntervalEnumerator` object is not really part of the enclosing `Interval` object. The `IntervalEnumerator` can, however, access (both public and private) class variables (static variables) of class `Interval`.

We could as well have placed the class `IntervalEnumerator` outside the class `Interval`, simply as a sibling class of `Interval`. But class `IntervalEnumerator` would just pollute the enclosing namespace. The `IntervalEnumerator` is only relevant inside the interval. Therefore we place it as a member of class `Interval`. By making it private we, furthermore, prevent clients of class `Interval` to access it.

Nested classes are, in general, a more advanced topic. It has, in part, something to do with scoping rules in relation to the outer classes, and in relation to superclasses. Java is more sophisticated than C# in its support of nested classes. In java, an inner class `I` in the surrounding class `C` is a nested class for which instances of `I` is connected to (is part of) a particular instance of `C`. See also our discussion of Java in relation to C# in Section 7.3.

```
using System;
using System.Collections;

public class app {

  public static void Main(){

    Interval iv1 = new Interval(14,17);

    foreach(int k in iv1){
      Console.Write("{0,4}", k);
    }
    Console.WriteLine();

    IEnumerator e = iv1.GetEnumerator();
    while (e.MoveNext()){
      Console.Write("{0,4}", (int)e.Current);
    }
    Console.WriteLine();
  }

}
```

Program 31.10 *Iteration with and without foreach based on the enumerator.*

31.7. Sample use of IFormattable

Lecture 8 - slide 17

The `IFormattable` interface prescribes a `ToString` method of two parameters. As such, the `ToString` method of `IFormattable` is different from the well-known `ToString` method of class `Object`, which is parameterless, see Section 28.3. Both methods produce a text string. The new `ToString` method is used when we need more control of the textual result.

Here follows a reproduction of `IFormattable` from the `System` namespace.

```
using System;

public interface IFormattable{
  string ToString(string format, IFormatProvider formatProvider);
}
```

Program 31.11 *A reproduction of the interface IFormattable.*

We can characterize the `ToString` method in the following way:

- The first parameter is typically a single letter formatting string, and the other is an `IFormatProvider`
- The `IformatProvider` can provide culture sensible information.
- `ToString` from `Object` typically calls `ToString(null, null)`

The first parameter of `ToString` is typically a string with a single character. For simple types as well as `DateTime`, a number of predefined formatting strings are defined. We have seen an example in Section 6.10. For the types we program we can define our own formatting letters. This is known as *custom formatting*. Below, in Program 31.12 we will show how to program custom formatting of a playing card struct.

The second parameter of `ToString` is of type `IFormatProvider`, which is another interface from the `System` namespace. An object of type `IFormatProvider` typically provides culture sensible formatting information. For simple types and for `DateTime`, a format provider represents details such as the currency symbol, the decimal point symbol, or time-related formatting symbols. If the second parameter is `null`, the object bound to `CultureInfo.CurrentCulture` should be used as the default format provider.

Below we show how to program custom formatting of struct `Card`, which we first met in the context of structs in Section 14.3. Notice that struct `Card` implements `Iformattable`. The details in the two `Tostring` methods should be easy to understand.

```
using System;

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

public struct Card: IFormattable{
  private CardSuite suite;
  private CardValue value;

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

  // Card methods and properties here...

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

  public override String ToString(){
    return this.ToString(null, null);
  }

  public String ToString(string format, IFormatProvider fp){
    if (format == null || format == "G" || format == "L")
        return String.Format("Card Suite: {0}, Value: {1}, Color: {2}",
                             suite, value, Color().ToString());

    else if (format == "S")
        return String.Format("Card {0}: {1}", suite, (int)value);

    else if (format == "V")
        return String.Format("Card value: {0}", value);

    else throw new FormatException(
                     String.Format("Invalid format: {0}", format));
  }

}
```

Program 31.12 *The struct Card that implements IFormattable.*

In Program 31.13 we show how to make use of custom formatting of playing card objects. The resulting output can be seen in Listing 31.14.

```
using System;

class CardClient{

  public static void Main(){
    Card c1 = new Card(CardSuite.Hearts, CardValue.Eight),
         c2 = new Card(CardSuite.Diamonds, CardValue.King);

    Console.WriteLine("c1 is a {0}", c1);
    Console.WriteLine("c1 is a {0:S}", c1); Console.WriteLine();

    Console.WriteLine("c2 is a {0:S}", c2);
    Console.WriteLine("c2 is a {0:L}", c2);
    Console.WriteLine("c2 is a {0:V}", c2);


  }

}
```

Program 31.13 *A client of Card which applies formatting of cards.*

```
c1 is a Card Suite: Hearts, Value: Eight, Color: Color [Red]
c1 is a Card Hearts: 8

c2 is a Card Diamonds: 13
c2 is a Card Suite: Diamonds, Value: King, Color: Color [Red]
c2 is a Card value: King
```

Listing 31.14 *Output from the client program.*

31.8. Explicit Interface Member Implementations

Lecture 8 - slide 18

Interfaces give rise to multiple inheritance, and therefore we need to be able to deal with the challenges of multiple inheritance. These have already been discussed in Section 27.5.

The problems, as well as the C# solution, can be summarized in the following way:

> If a member of an interface collides with a member of a class, the member of the interface can be implemented as an explicit interface member
>
> Explicit interface members can also be used to implement several interfaces with colliding members

The programs shown below illustrate the problem and the solution. The class `Card`, in Program 31.15 has a `Value` property. The interface `IGameObject` in Program 31.16 also prescribes a `Value` property. (It is similar to the interface of Program 31.1 which we have encountered earlier in this chapter). When class `Card` implements `IGameObject` in Program 31.17 the new version of class `Card` will need to distinguish between its own `Value` property and the `Value` property it implements because of the interface `IGameObject`. How can this be done?

The solution to the problem is called *explicit interface member implementation*. In line 30-32 of Program 31.17, emphasized in **purple**, we use the `IgameObject.Value` syntax to make it clear that here we implement the `Value` property from `IGameObject`. This is an explicit interface implementation.

In the client classes of class `Card` we need access to both `Value` operations. In order to access the explicit interface implementation of `Value` from the `Card` variable `cs` (declared in line 6) we need to cast `cs` to the interface `IGameObject`. This is illustrated in line 14 of Program 31.18. The output of Program 31.18 in Listing 31.19 reveals that everything works as expected.

```
using System;

public class Card{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  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 31.15 *The class Playing card with a property Value.*

```
public enum GameObjectMedium {Paper, Plastic, Electronic}

public interface IGameObject{

  int Value{
    get;
  }

  GameObjectMedium Medium{
    get;
  }
}
```

Program 31.16 *The Interface IGameObject with a conflicting Value property.*

```
using System;

public class Card: IGameObject{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  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);
  }

  int IGameObject.Value{
    get { return (int)(this.value); }
  }

  public GameObjectMedium Medium{
    get{
      return GameObjectMedium.Paper;
    }
  }
}
```

Program 31.17 *A class Card which implements IGameObject.*

```
using System;

class Client{

  public static void Main(){
    Card cs =
       new Card(Card.CardSuite.spades, Card.CardValue.queen);

    // Use of Value from Card
    Console.WriteLine(cs.Value);

    // Must cast to use the implementation of
    // Value from IGameObject
    Console.WriteLine(((IGameObject)cs).Value);
  }
}
```

Program 31.18 *Sample use of class Card in a Client class.*

```
queen
12
```

Listing 31.19 *Output of Card Client.*

In some situations, an explicit interface implementation can also be used to "hide" an operation that we are forced to implement because the interface requests it. We will meet an example in Section 45.14, where we want to make it difficult to use the `Add` operation on a linked list. Another example is presented in the context of dictionaries in Section 46.3.

31.9. References

[Hejlsberg06]	Anders Hejlsberg, Scott Wiltamuth and Peter Golde, *The C# Programming Language*. Addison-Wesley, 2006.
[Ecma-334]	"The C# Language Specification", June 2005. ECMA-334.

32. Patterns and Techniques

This chapter is the last one in our second lecture about inheritance. The chapter is about patterns and programming techniques related to inheritance. Similar chapters appeared in Chapter 16 and Chapter 24 for classes/objects and for operations respectively.

32.1. The Composite design pattern

Lecture 8 - slide 20

The ***Composite*** design pattern, which we are about to study, is probably the most frequently occurring GOF design pattern at all. Most real-life programs that we write benefit from it. Recall from Section 16.2 that the GOF design patterns are the ones described in the original design pattern book [Gamma96].

A ***Composite*** deals with hierarchical structures of objects. In more practical terms, the pattern deals with tree-tructures whose nodes are objects. The main idea behind the pattern is to provide a *uniform interface* to both leaves and inner nodes in the tree.

From a client point of view, it is easy to operate on the nodes of a ***Composite***. The reason is that all participating objects share the interface provided by the abstract *`Component`* class.

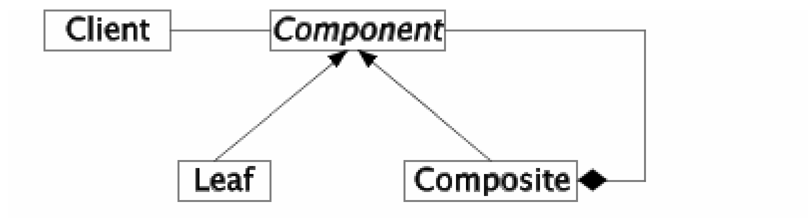

Figure 32.1 *A template of the class structure in the Composite design pattern.*

In Figure 32.1 we show the three classes that - at the principled level - make up a ***Composite:*** The abstract class *`Component`* and its two subclasses `Leaf` and `Composite`. The important things to notice are:

- The diagram in Figure 32.1 is a class diagram, not an object diagram.
- Clients access both `Leaf` nodes and `Composite` nodes (inner nodes in the tree) via the interface provided by the abstract class *`Component`* .
- The `Composite` (inner) node aggregates one or more `Components` , either `Leaf` nodes or (recursively) other `Composite` nodes. This makes up the tree structure. It is important the you are able to grasp the idea that the aggregation in Figure 32.1 gives rise to a recursive tree structure of objects.

In the following sections we will study an example of a composite design pattern which allows us to represent songs of notes and pauses. In appendix Section 58.3 we discuss another example, involving a sequence of numbers and the type `Interval`.

The tree structure may be non-mutable and built via constructors Alternatively, the tree structure may be mutable, and built via `Add` and `Remove` operations

32.2. A Composite Example: Music Elements

Lecture 8 - slide 21

The example in this section stems from the mini project programming (MIP) exam of January 2008 [mip-jan-08]. Imagine that we are interested in a representation of music in terms of notes and pauses. Such a representation can - in a natural way - be described as a ***Composite***, see Figure 32.2. In this composite structure, both a `Note` and a `Pause` are `MusicElements`. A `SequentialMusicElement` consists of a number of `MusicElements`, such as `Notes`, `Pauses`, and other `MusicElements`. The immediate constituents of a `SequentialMusicElement` are played sequentially, one after the other. A `ParallelMusicElement` is composed similar to `SequentialMusicElement`. The immediate constituents of a `ParallelMusicElement` are played at the same time, however.

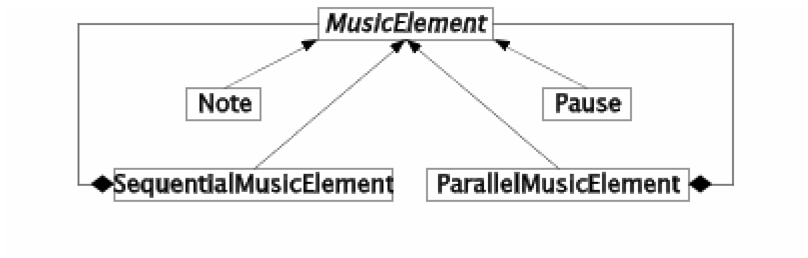

Figure 32.2 *The class diagram of Music Elements*

As we will see in Program 32.3 a `Note` is characterized by a duration, value, volume, and instrument. A `Pause` is characterized by a duration. As such, it may make sense to have a common superclass of `Note` and `Pause`. In the same way, it may be considered to have a common superclass of `SequentialMusicElement` and `ParallelMusicElement` which captures their common aggregation of `MusicElements`.

A number of different operations can be applied uniformly on all `MusicElements`: `Play`, `Transpose`, `TimeStretch`, `NewInstrument`, `Fade`, etc. Below, in Program 32.3 we program the operations `Linearize`, `Duration`, and `Transpose`. The `Linearize` operations transforms a music element to a sequence of lower-level objects which represent MIDI events. A sequence of MIDI events can be played on most computers. In this way, `Linearize` becomes the indirect `Play` operation.

32.3. An application of Music Elements

Lecture 8 - slide 22

As we already realized in Section 32.1 the objects in a ***Composite*** are organized in a tree structure. In Figure 32.3 we show an example of a `SequentialMusicElement`. When we play the `SequentialMusicElement` in Figure 32.3 we will first hear note N1. After N1 comes a pause P followed by the notes N2 and N3. Following N3 we will hear N4, N5 and N6 which are all played simultaneously. As such, N4-N6 may form a musical chord. In the web edition of the material we link to a MIDI file of a structure similar to Figure 32.3 [midi-sample].

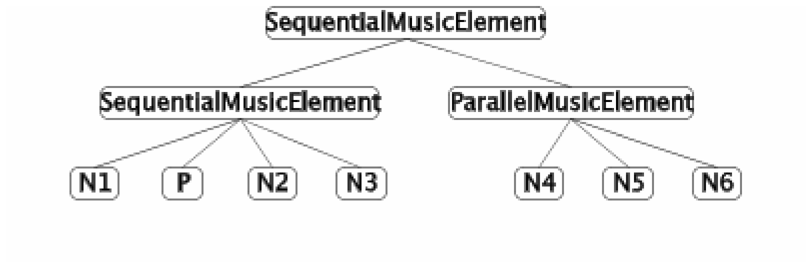

Figure 32.3 *A possible tree of objects which represent various music elements. Nodes named Ni are `Note` instances, and the node named P is a `Pause` instance*

Below, in Program 32.1 we show a program that creates a `SequentialMusicElement` similar to the tree-structure drawn in Figure 32.3 The program relies on the auxiliary class `Song`. The class `Song` and another supporting class `TimedNote` are available to interested readers [song-and-timednote-classes]. Using these two classes it is easy to generate MIDI files from `MusicElement` objects.

```
public class Application{

  public static void Main(){

    MusicElement someMusic =
     SequentialMusicElement.MakeSequentialMusicElement(
       SequentialMusicElement.MakeSequentialMusicElement(
         new Note(60, 480),
         new Pause(480),
         new Note(64, 480),
         new Note(60, 480)),
       ParallelMusicElement.MakeParallelMusicElement(
         new Note(60, 960),
         new Note(64, 960),
         new Note(67, 960)
       ));

    Song aSong = new Song(someMusic.Linearize(0));
    aSong.WriteStandardMidiFile("song.mid");
  }
}
```

Program 32.1 *An application of some MusicElement objects.*

32.4. Implementation of MusicElement classes

Lecture 8 - slide 23

In this section we show an implementation of the `MusicElement` classes of Figure 32.2. The classes give rise to non-mutable objects, along the lines of the discussion in Section 12.5.

We start by showing the abstract class `MusicElement`, see Program 32.2. It announces the property `Duration`, the method `Transpose`, and the method `Linearize`. Other of the mentioned music-related operations are not included here. As you probably expect, `Duration` returns the total length of a `MusicElement`. `Transpose` changes the value (the pitch) of a `MusicElement`. `Linearize` transforms a `MusicElement` to an array of (lower-level) `TimeNote` objects [song-and-timednote-classes].

```
public abstract class MusicElement{

  public abstract int Duration{
    get;
  }

  public abstract MusicElement Transpose(int levels);

  public abstract TimedNote[] Linearize(int startTime);
}
```

Program 32.2 *The abstract class MusicElement.*

The class `Note` is shown next, see Program 32.3. `Note` encapsulates the note value, duration, volume, and instrument (see line 5-8). Following two constructors, we see the property `Duration` which simply returns the value of the instance variable `duration`. The method `Linearize` carries out the transformation of the `Note` to a singular array of `TimedNote`. The `Transpose` method adds to the value of the `Note`. The shown activation of `ByteBetween` enforces that the value is between 0 and 127.

```
1  using System;
2  
3  public class Note: MusicElement{
4  
5    private byte value;
6    private int duration;
7    private byte volume;
8    private Instrument instrument;
9  
10   public Note(byte value, int duration, byte volume,
11               Instrument instrument){
12     this.value = value;
13     this.duration = duration;
14     this.volume = volume;
15     this.instrument = instrument;
16   }
17 
18   public Note(byte value, int duration):
19    this(value, duration, 64, Instrument.Piano){
20   }
21 
22   public override int Duration{
23     get{
24       return duration;
25     }
26   }
27 
28   public override TimedNote[] Linearize(int startTime){
29     TimedNote[] result = new TimedNote[1];
30     result[0] = new TimedNote(startTime, value, duration, volume,
31                               instrument);
32     return result;
33   }
34 
35   public override MusicElement Transpose(int levels){
36      return new Note(Util.ByteBetween(value + levels, 0, 127),
37                      duration, volume, instrument);
38   }
39 }
```

Program 32.3 *The class Note.*

The class `Pause` shown in Program 32.4 is almost trivial.

```
using System;

public class Pause: MusicElement{

  private int duration;

  public Pause(int duration){
    this.duration = duration;
  }

  public override int Duration{
    get{
      return duration;
    }
  }

  public override TimedNote[] Linearize(int startTime){
    return new TimedNote[0];
  }

  public override MusicElement Transpose(int levels){
     return new Pause(this.Duration);
  }
}
```

Program 32.4 *The class Pause.*

The class `SequentialMusicElement` represents the sequence of `MusicElements` as a list of type `List<T>`. Besides the constructor, `SequentialMusicElement` offers a *factory method* for convenient creation of an instance. Factory methods have been discussed in Section 16.4. Program 32.1 shows how the factory method can be applied. `Duration` adds the duration of the `MusicElement` parts together. Notice that this may cause recursive addition. Likewise, `Transpose` carries out recursive transpositions of the `MusicElement` parts.

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

public class SequentialMusicElement: MusicElement{
  private List<MusicElement> elements;

  public SequentialMusicElement(MusicElement[] elements){
    this.elements = new List<MusicElement>(elements);
  }

  // Factory method:
  public static MusicElement
    MakeSequentialMusicElement(params MusicElement[] elements){
      return new SequentialMusicElement(elements);
  }

  public override TimedNote[] Linearize(int startTime){
    int time = startTime;
    List<TimedNote> result = new List<TimedNote>();

    foreach(MusicElement me in elements){
      result.AddRange(me.Linearize(time));
      time = time + me.Duration;
    }

    return result.ToArray();
  }

  public override int Duration{
    get{
      int result = 0;

      foreach(MusicElement me in elements){
        result += me.Duration;
      }

      return result;
    }
  }

  public override MusicElement Transpose(int levels){
    List<MusicElement> transposedElements = new List<MusicElement>();

    foreach(MusicElement me in elements)
      transposedElements.Add(me.Transpose(levels));

    return new SequentialMusicElement(transposedElements.ToArray());
  }
}
```

Program 32.5 *The class SequentialMusicElement.*

The class `ParallelMusicElement` resembles `SequentialMusicElement` a lot. Notice, however, the different implementation of `Duration` in line 29-39.

```
1  using System;
2  using System.Collections.Generic;
3  
4  public class ParallelMusicElement: MusicElement{
5    private List<MusicElement> elements;
6  
7    public ParallelMusicElement(MusicElement[] elements){
8      this.elements = new List<MusicElement>(elements);
9    }
10 
11   // Factory method:
12   public static MusicElement
13     MakeParallelMusicElement(params MusicElement[] elements){
14       return new ParallelMusicElement(elements);
15   }
16 
17   public override TimedNote[] Linearize(int startTime){
18     int time = startTime;
19     List<TimedNote> result = new List<TimedNote>();
20 
21     foreach(MusicElement me in elements){
22       result.AddRange(me.Linearize(time));
23       time = startTime;
24     }
25 
26     return result.ToArray();
27   }
28 
29   public override int Duration{
30     get{
31       int result = 0;
32 
33       foreach(MusicElement me in elements){
34         result = Math.Max(result, me.Duration);
35       }
36 
37       return result;
38     }
```

```
  }

  public override MusicElement Transpose(int levels){
    List<MusicElement> transposedElements = new List<MusicElement>();

    foreach(MusicElement me in elements)
      transposedElements.Add(me.Transpose(levels));

    return new ParallelMusicElement(transposedElements.ToArray());
  }
}
```

Program 32.6 *The class ParallelMusicElement.*

This completes our discussion of the `MusicElement` composite. The important things to pick up from the example are:

1. The tree structure of objects defined by the subclasses of `MusicElement` .
2. The uniform interface of music-related operations provided to clients of `MusicElement` .

As stressed in Section 32.1 these are the primary merits of ***Composite***.

In Section 58.3 of the appendix we present an additional and similar example of a composite which involves an `Interval`. `Interval` is the type we encountered in Section 21.3 when we discussed operator overloading.

32.5. A Composite Example: A GUI

Lecture 8 - slide 27

We will study yet another example of a ***Composite*** design pattern. A graphical user interface (GUI) is composed of a number of *forms*, such as buttons and textboxes. The classes behind these forms make up a ***Composite*** design pattern.

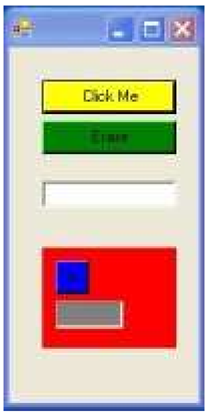

Figure 32.4 *A Form (Window) with two buttons, a textbox, and a panel.*

We construct the simple GUI depicted in Figure 32.4. The actual hierarchy of objects involved are shown in Figure 32.5. Thus, the GUI is composed of three buttons (yellow, green, and blue) and two textboxes (white and grey). The blue button and the grey textbox are aggregated into a so-called panel (which has red background in Figure 32.4).

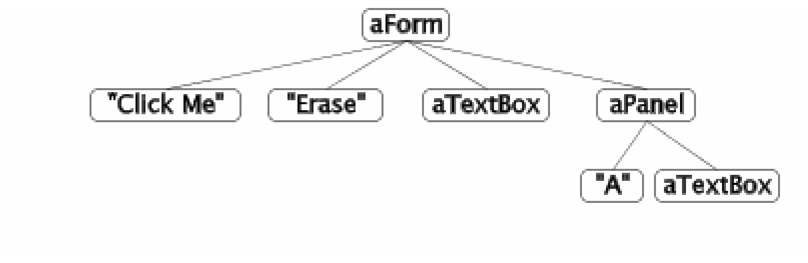

Figure 32.5 *The tree of objects behind the graphical user interface. These objects represents a composite design pattern in an executing program.*

The Form class hierarchy of .NET and C# is very large. A small extract is shown in Figure 32.6. Apart from class `Component`, all classes are from the namespace `System.Windows.Forms`.

There are two ***Composites*** in Figure 32.6. The first one is (object) rooted by class `Form`, which may aggregate an arbitrary number of Windows form objects. The class `Form` represents a window. The class `Control` is the superclass of GUI elements that displays information on the screen. There are approximate 25 immediate and direct subclasses of class `Control`. In reality the classes `TextBox`, `Button`, and `Panel` are all indirect subclasses of `Control`.

The other Composite is, symmetrically, (object) rooted by `Panel`, which like `Form` may aggregate an arbitrary number of `Form` objects. Class `Pane` is intended for grouping of a collection of controls.

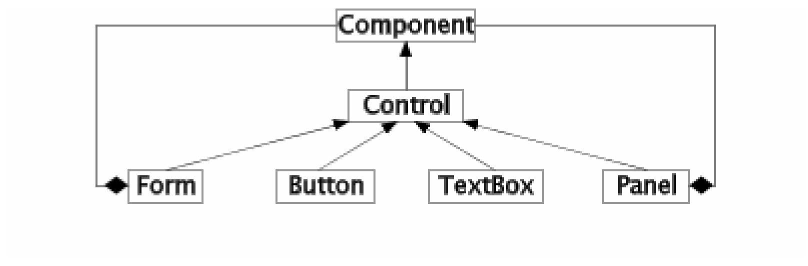

Figure 32.6 *An extract of the Windows Form classes for GUI building. We see two Composites among these classes.*

Below, in Figure 32.6 we show how to construct the form object tree shown in Figure 32.5, which gives rise to the GUI of Figure 32.4. We program a class which we name `Window`. Our `Window` class inherits from class `Form`. Thus, our `Window` ***is a*** `Form`. Shown in **blue** we highlight instantiation of GUI elements. Shown in **purple** we highlight the actual construction of the tree structure of Figure 32.5. The `Controls` property of a Form, referred in line 60 - 67, give access to a collection of controls, of type `ControlCollection`.

As it appears in line 23 and 31, we also add a couple of event handlers, programmed as private methods from line 70 - 83. We have discussed event handlers in Chapter 23. The associated event handlers just acknowledge when we click on of the three buttons of the GUI.

```
1  using System;
2  using System.Windows.Forms;
3  using System.Drawing;
4  
5  // In System:
6  // public delegate void EventHandler (Object sender, EventArgs e)
7  
8  public class Window: Form{
9  
10   Button b1, b2, paBt;
11   Panel pa;
```

```
  TextBox tb, paTb;

  // Constructor
  public Window (){
    this.Size=new Size(150,300);

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

    pa = new Panel();
    pa.Location = new Point(25,150);
    pa.Size=new Size(100, 75);
    pa.BackColor=Color.Red;

    paBt = new Button();
    paBt.Text="A";
    paBt.Location = new Point(10,10);
    paBt.Size=new Size(25,25);
    paBt.BackColor=Color.Blue;
    paBt.Click += PanelButtonClickHandler;

    paTb = new TextBox();
    paTb.Location = new Point(10,40);
    paTb.Size=new Size(50,25);
    paTb.BackColor=Color.Gray;
    paTb.ReadOnly=true;
    paTb.RightToLeft=RightToLeft.Yes;

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

    pa.Controls.Add(paBt);
    pa.Controls.Add(paTb);

    this.Controls.Add(pa);
  }

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

  // Eventhandler:
  private void PanelButtonClickHandler(object obj, EventArgs ea) {
    paTb.Text += "A";
  }

  // Eventhandler:
  private void EraseHandler(object obj, EventArgs ea) {
    tb.Text = "";
  }

}

class ButtonTest{

  public static void Main(){
    Window win = new Window();
    Application.Run(win);
  }

}
```

Program 32.7 *A program that builds a sample composite graphical user interface.*

32.6. Cloning

Lecture 8 - slide 30

We briefly discussed copying of objects in Section 13.4 of the lecture about classes and objects. In this section we will continue this discussion. First we will distinguish between different types of object copying. Later, in Section 32.7, we will see how to enable the pre-existing `MemberwiseClone` operation to client classes.

Instead of the word "copy" we often use the word "clone":

Cloning creates a copy of an existing object

There are different kinds of cloning, distinguished by the copying depth:

- **Shallow cloning**:
  - Instance variables of value type: Copied bit-by-bit
  - Instance variables of reference types:
    - The reference is copied
    - The object pointed at by the reference is **not** copied
- **Deep cloning**:
  - Like shallow cloning
  - But objects referred by references are copied recursively

Shallow cloning is the variant supported by the `MemberwiseClone` operation in Section 32.7. Only a single object is copied.

Deep cloning copies a network of objects, and it may, in general, involve many objects.

Recall that cloning is only relevant for instances of classes, for which reference semantics apply (see Chapter 13). Values of structs obey value semantics, and as such struct values are (shallow) copied by assignments and by parameter passing. See Chapter 14 for additional details.

32.7. Cloning in C#

Lecture 8 - slide 31

Shallow cloning is supported "internally" by any object in a C# program execution. The reason is that any object inherit from class `Object` in which the protected method `MemberwiseClone` implements shallow cloning. (See Section 28.3 for an overview of the methods in class `Object`). Recall from Section 27.3 that a protected method of a class C is visible in C and in the subclasses of c, but not in clients of C.

In this section we will see how we can unleash the protected `MemberwiseClone` operation as a public operation of an arbitrary class.

Below, in Program 32.8 we show how to implement a cloneable `Point` class. First, notice that `Point` implements the interface `ICloneable`, which prescribes a single method called `Clone`. We have already in Section 31.4 seen `ICloneable` in the context of other flavoring interfaces from the C# libraries. The public method `Clone` of class `Point`, shown in **purple**, delegates its work to the protected method `MemberwiseClone`. In other words, our `Clone` methods send a `MemberwiseClone` message to the current `Point` object. `MemberWiseClone` makes the bitwise, shallow copy of the point, and it returns it. Notice that from a static point of view, the returned object is of type `Object`. As we will see below, this will typically imply a need to cast the returned object to a `Point`.

Although a `Clone` method typically delegates its work to `MemberwiseClone`, it is not necessary to do so. `Clone` may, alternatively, use a constructor and appropriate object mutations in order to produce the copy, which makes sense for the class in question (which is class `Point` in the example shown below).

```
using System;

public class Point: ICloneable {
  private double x, y;

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

  public double X {
    get {return x;}
    set {x = value;}
  }

  public double Y {
    get {return y;}
    set {y = value;}
  }

  public Point move(double dx, double dy){
    Point result = (Point)MemberwiseClone();  // cloning from within Point is OK.
    result.x = x + dx;
    result.y = y + dy;
    return result;
  }

  // public Clone method that delegates the work of
  // the protected method MemberwiseClone();
  public Object Clone(){
    return MemberwiseClone();
  }

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

Program 32.8 *A cloneable class Point.*

In Program 32.9 we show how a client of class `Point` uses the implemented `Clone` operation. Notice the casting, and notice that the subexpression `p1.Clone()` is evaluated before the casting. (A possible misconception would be that `(Point)p1` is evaluated first). The evaluation order is due to the precedence of the cast operator in relation to the precedence of the dot operator, see Table 6.1.

```
using System;

public class Application{

  public static void Main(){
    Point p1 = new Point(1.1, 2.2),
          p2, p3;

    p2 = (Point)p1.Clone();  // First p1.Clone(), then cast to Point.
    p3 = p1.move(3.3, 4.4);
    Console.WriteLine("{0} {1} {2}", p1, p2, p3);
  }

}
```

Program 32.9 *A sample client of class Point.*

It may be tempting to circumvent the `ICloneable` interface, the implementation of our own clone operation, and delegation to `MemberwiseClone`. This temptation is illustrated in Program 32.10. The compiler will find out, and it tells that we cannot just call `MemberwiseClone`, because it is not a public operation.

Why make life so difficult? Why not support shallow copying of all objects in an easy way, by making `MemberwiseClone` public in class `Object`? The reason is that the designers of the programming language (C#, and Java as well) have decided that the programmer of a class should make an explicit decision about which objects should be cloneable.

There are almost certainly some classes for which we do not want copying of instances, singletons (see Section 16.3) for instance. There are also some classes in which we do not want the standard bitwise copying provided by `MemberwiseClone`. Such classes should behave like Program 32.8 shown above, but instead of delegating the cloning to `MemberwiseClone`, the copy operation should be programmed in the `Clone` method to suit the desired copying semantics.

```
using System;

public class Application{

  public static void Main(){
    Point p1 = new Point(1.1, 2.2),
          p2, p3;

    p2 = (Point)p1.MemberwiseClone();
    // Compile-time error.
    // Cannot access protected member 'object.MemberwiseClone()'
    // via a qualifier of type 'Point'

    p3 = p1.move(3.3, 4.4);
    Console.WriteLine("{0} {1} {2}", p1, p2, p3);
  }

}
```

Program 32.10 *Illegal cloning with MemberwiseClone.*

32.8. Cloning versus use of copy constructors

Lecture 8 - slide 32

In Section 32.7 we found out that cloning of class instances - on purpose - is rather cumbersome in C#. Therefore we have earlier recommended the use of *copy constructors* as an alternative means. See Section 12.5 for details and for an example.

In this section we will evaluate and exemplify the power of copying by cloning (as in Section 32.7) relative to copying by use of copy constructors.

In a nutshell, the insight can be summarized in this way:

> Cloning with **`obj.Clone()`** is more powerful than use of copy constructors, because **`obj.Clone()`** may exploit polymorphism and dynamic binding

In order to illustrate the differences between cloning (by use of the `clone` method) and copying (by use of a copy constructor) we will again use the class `Point`. Below, in Section 32.7 we show a version similar to Program 32.8 but now with an additional copy constructor (line 12 - 14).

```
1  using System;
2  using System.Drawing;
3
4  public class Point: ICloneable {
5    protected double x, y;
6
7    public Point(double x, double y){
8     this.x = x; this.y = y;
9    }
10
11   // Copy constructor
12   public Point(Point p){
13    this.x = p.x; this.y = p.y;
14   }
15
16   public virtual double X {
17     get {return x;}
18     set {x = value;}
19   }
20
21   public virtual double Y {
22     get {return y;}
23     set {y = value;}
```

```
  }

  public virtual Point move(double dx, double dy){
    Point result = (Point)MemberwiseClone();  // cloning from within Point is OK.
    result.x = x + dx;
    result.y = y + dy;
    return result;
  }

  // public Clone method that delegates the work of
  // the protected method MemberwiseClone();
  public virtual Object Clone(){
    return MemberwiseClone();
  }

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

Program 32.11 *A cloneable class Point.*

We also show a subclass of `Point` called `ColorPoint`, see Program 32.12. It adds a `color` instance variable to the instance variables inherited from class `Point`, and it has its own copy constructor in line 10 - 13.

```
1  public class ColorPoint: Point{
2    protected Color color;
3
4    public ColorPoint(double x, double y, Color c):
5        base(x,y){
6      this.color = c;
7    }
8
9    // Copy constructor
10   public ColorPoint(ColorPoint cp):
11       base(cp.x,cp.y){
12     this.color = cp.color;
13   }
14
15   // Clone method is inherited
16
17   public override string ToString(){
18     return "ColorPoint: " + "(" + x + "," + y + ")" + ":" + color;
19   }
20 }
```

Program 32.12 *A cloneable class ColorPoint.*

In the `Color` and `ColorPoint` client program, shown below in Program 32.13, you should focus on the list `pointList`, as declared in line 14. We add two `Point` objects and two `ColorPoint` objects to `pointList` in line 17 - 20. Next, in the foreach loop starting at line 23 we clone each of the four points in the list, and we add the cloned points to the initially empty list `clonedPointList`. The elements in `clonedPointList` are printed at the end of the program. The output - see Listing 32.14 - reveals that the cloning is successful. We end up having two `Point` instances and two `ColorPoint` instances in `clonedPointList`.

```
1  using System;
2  using System.Drawing;
3  using System.Collections.Generic;
4
5  public class Application{
6
```

```
  public static void Main(){
    Point p1 =       new Point(1.1, 2.2),
          p2 =       new Point(3.3, 4.4);

    ColorPoint cp1 = new ColorPoint(5.5, 6.6, Color.Red),
               cp2 = new ColorPoint(7.7, 8.8, Color.Blue);

    List<Point> pointList = new List<Point>(),
                clonedPointList = new List<Point>();

    pointList.Add(p1);
    pointList.Add(cp1);
    pointList.Add(p2);
    pointList.Add(cp2);

    // Clone the points in pointList and add them to clonedPointList:
    foreach(Point p in pointList){
      clonedPointList.Add((Point)(p.Clone()));
    }

    foreach(Point p in clonedPointList)
      Console.WriteLine("{0}", p);

  }
}
```

Program 32.13 *Polymorphic Cloning of Points.*

```
Point: (1,1,2,2).
ColorPoint: (5,5,6,6):Color [Red]
Point: (3,3,4,4).
ColorPoint: (7,7,8,8):Color [Blue]
```

Listing 32.14 *Output of both polymorphic and non-polymorphic cloning.*

Let us now attempt to replicate the functionality of Program 32.13 with use of copy constructors, see Program 32.15. The attempt, shown in Program 32.15 in line 24 - 26 fails, because the activation of the copy constructors deliver `Point` objects, even if a `ColorPoint` object is passed as input. Instead we must perform explicit type dispatch, as shown in line 29-34. Clearly, constructors cannot exhibit virtual behavior.

The solution in Program 32.13 based on the `Point` and `ColorPoint` classes in Program 32.11 and Program 32.12 works because the `Clone` method in Program 32.11 (line 35 - 37) is inherited by `ColorPoint`. As already explained, the inherited method delegates its work to `MemberwiseClone`, which always copies its receiver. Thus, if `MemberwiseClone` is activated on a `ColorPoint` object it copies a `ColorPoint` object.

```
1  using System;
2  using System.Drawing;
3  using System.Collections.Generic;
4  
5  public class Application{
6  
7    public static void Main(){
8      Point p1 =       new Point(1.1, 2.2),
9            p2 =       new Point(3.3, 4.4);
10 
11     ColorPoint cp1 = new ColorPoint(5.5, 6.6, Color.Red),
12                cp2 = new ColorPoint(7.7, 8.8, Color.Blue);
13 
14     List<Point> pointList = new List<Point>(),
15                 clonedPointList = new List<Point>();
16 
```

```
    pointList.Add(p1);
    pointList.Add(cp1);
    pointList.Add(p2);
    pointList.Add(cp2);

//  Cannot copy ColorPoint objects with copy constructor of Point.
//  Compiles and runs, but gives wrong result.
//  foreach(Point p in pointList){
//    clonedPointList.Add(new Point(p));
//  }

    // Explicit type dispatch:
    foreach(Point p in pointList){
      if (p is ColorPoint)
        clonedPointList.Add(new ColorPoint((ColorPoint)p));
      else if (p is Point)
        clonedPointList.Add(new Point(p));
    }

    foreach(Point p in clonedPointList)
      Console.WriteLine("{0}", p);

  }
}
```

Program 32.15 *Non-polymorphic Cloning of Points - with use of copy constructors.*

32.9. The fragile base class problem

Lecture 8 - slide 33

As the next part of this Pattern and Techniques chapter we will study the so-called fragile base class problem.

The problem can be summarized in this way:

If all methods are virtual it is possible to introduce erroneous dynamic bindings

This can happen if a new method in a superclass is given the same name as a dangerous method in a subclass

The program in Program 32.16 is a principled ABC example. `B` is a subclass of `A`, and `B` has a dangerous method `M2`. As a dangerous method, clients of class `B` must be fully aware that `M2` is called, because it can have serious consequences. (A possible serious consequence may be to erase the entire harddisk). As illustrated in the `Client` class, `M2` can only be activated on a `B` object.

```
// Original program. No problems.

using System;

class A {

  public void M1(){
    Console.WriteLine("Method 1");
  }
}

```

```
class B: A {

  public void M2(){
    Console.WriteLine("Dangerous Method 2");
  }
}

class Client{

  public static void Main(){
    A a = new B();
    B b = new B();

    a.M1();  // Nothing dangerous expected
//  a.M2();  // Compile-time error
             // 'A' does not contain a definition for 'M2'
    b.M2();  // Expects dangerous operation
  }
}
```

Program 32.16 *The initial program.*

Let us now assume that we replace class `A` with a new version with the following changes:

1. A new virtual method `M2` is added to A.
2. The existing method `M1` in `A` now calls `M2`

This is shown in Program 32.17.

We will, in addition, assume that all involved methods (`M1` and `M2`) are virtual, and that `M2` in `B` overrides `M2` in `A`. In C# this is not a natural assumption, but in Java this is the default semantics (and the only possible semantics).

It is purely accidental that the new method in class `A` has the same name as the dangerous method `M2` in class `B`.

In the `Client` class in Program 32.17 `a.M1()` will - unexpectedly - call the dangerous method `M2` in class `B`, because the variable `a` has dynamic type `B`. Similarly, `a.M2()` calls `M2` in `B`. The programmer, who wrote class `A`, expected that the expression `a.M1()` would call the sibling method `M2` in class `A`! This could come as an unpleasant surprise.

```
// Dangerous program.
// M2 is virtual in A and overridden in B.
// Compiles and runs
// Default Java semantics.

using System;

// New version of A
class A {

  public void M1(){
    Console.WriteLine("Method 1");
    this.M2();
  }

  // New method in this version.
  // Same name as the dangerous operation in subclass B
  public  virtual  void M2(){
    Console.WriteLine("M2 in new version of A");
  }

}

class B: A {

  public override void M2(){
    Console.WriteLine("Dangerous Method 2");
  }
}

class Client{

  public static void Main(){
    A a = new B();
    B b = new B();

    a.M1();  // Nothing dangerous expected
             // Will, however, call the dangerous operation
             // because M2 is virtual.

    a.M2();  // Makes sense when M2 exists in class A.
             // Calls the dangerous method

    b.M2();  // Calls the dangerous method.
             // This is expected, however.
  }
}
```

Program 32.17 *The revised version with method A.M2 being virtual.*

```
Method 1
Dangerous Method 2
Dangerous Method 2
Dangerous Method 2
```

Listing 32.18 *Output of revised program.*

If we, in C#, just add the `M2` method to class `A`, and change `M1` such that it calls `M2`, as shown in Program 32.19 (only on web) it is not possible to compile class `B`. The problem is that we have a method, named `M2` in both class `A` and `B`. This is the problem that we have discussed in Section 28.9. The programmer should decide if `M2` in `B` should be declared as **`new`**, or if it should **`override`** `M2` from class `A`. In the latter case, `M2` in `A` must be declared as virtual.

If you want additional details about the fragile base class problem, the web-version of the paper contains two additional variants of Program 32.17.

The fragile base class problem illustrates a danger of using virtual methods (dynamic binding) all over the place. In rare situations, as the one constructed in Program 32.17, it may lead to dangerous results. To summarize, the problem arises if a method in a subclass is unintentionally called instead of a method in a superclass. In C#, both the superclass and the subclass must specify if dynamic binding should take place. In the superclass the involved method must be **`virtual`**, and in the subclass the method must use the **`override`** modifier. Alternatively, we may opt for static binding, as in Program 32.20. As illustrated by Program 32.19 the C# programmer is likely to get a warning in case he or she approaches the fragile base class problem.

32.10. Factory design patterns

Lecture 8 - slide 34

Instantiation of classes is done by the `new` operator (see Section 6.7) in cooperation with a constructor (see Section 12.4). Imagine that we need numerous instances of class `C` in a large program. This would lead to a situation where there appears may expressions of the form `new C(...)` in the program. Why can this be considered as a problem?

One problem with many occurrences of `new C(...)` is if we - eventually - would like to instantiate another class, say a subclass of `C`. In this situation we would prefer to make one change at a single place in the program, instead of a spread of changes throughout the program.

Another problem may occur if we need two or more constructors which we cannot easily distinguish by the formal parameters of the constructors. We have seen examples of such situations in Section 16.4.

As yet another problem, we may wish to introduce good names for object instantiations, beyond the possibilities of constructors.

Various uses of factory methods can be seen as solutions to the problems pointed out above. We will distinguish between the following variations of factory methods:

- Factory methods implemented with class methods (static methods) in C, or in another class
- The design pattern ***Factory Method*** which handles instantiation in instance methods of client subclasses
    - Relies on instance methods in class hierarchies with virtual methods
- The design pattern ***Abstract Factory*** which is good for instantiation of product families
    - Relies on instance methods in class hierarchies with virtual methods

As already pointed out, we have seen examples of static factory methods in Section 16.4. We will discuss the design pattern ***Factory Method*** below, in Section 32.11. In the current version of the material we do not discuss ***Abstract Factory***.

32.11. The design pattern Factory Method

Lecture 8 - slide 35

The ***Factory Method*** design pattern relies on virtual instance methods in a class hierarchy that take care of class instantiation. The ***Factory Method*** scene is shown diagrammatically in Figure 32.7 and programmatically in Program 32.22.

The problem is how to facilitate instantiation of different types of `Products` (line 3-13 in Program 32.22) in `SomeOperation` (line 20) of class `Creator`.

The Factory Method solution is to carry out the instantiation in overridden virtual methods in subclasses of class `Creator`. The actual instantiations take place in line 26 and 32 of Program 32.22. In `SomeOperation`, the highlighted call **`this.FactoryMethod()`** will either cause instantiation of `ConcreteProduct_1` or `ConcreteProduct_2`, depending on the dynamic type of the creator.

Subclasses of `Creator` decide which `Product` to instantiate

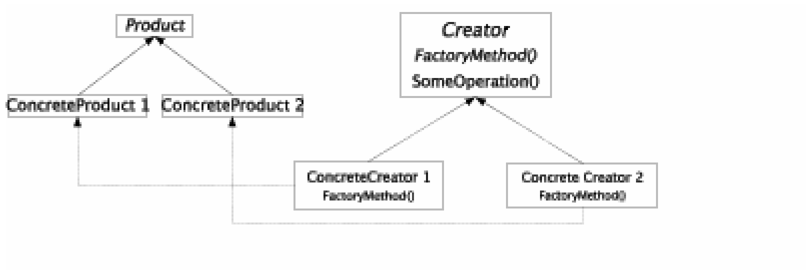

Figure 32.7 *A template of the class structure in the Factory Method design pattern.*

```
using System;

public abstract class Product{
  public Product(){}
}

public class ConcreteProduct_1: Product{
  public ConcreteProduct_1(){}
}

public class ConcreteProduct_2: Product{
  public ConcreteProduct_2(){}
}


public abstract class Creator{
  public abstract Product FactoryMethod();

  public void SomeOperation(){
    Product product =  FactoryMethod();
  }
}

public class ConcreteCreator_1: Creator{
  public override Product FactoryMethod(){
    return new ConcreteProduct_1();
  }
}

public class ConcreteCreator_2: Creator{
  public override Product FactoryMethod(){
    return new ConcreteProduct_2();
  }
}
```

Program 32.22 *Illustration of Factory Method in C#.*

Factory Method calls for a quite complicated scene of parallel class hierarchies. The key mechanism behind the pattern is the activation of a virtual method from a fully defined, non-abstract method in the (abstract) class `Creator`. In many contexts, ***Factory Method*** will be too complicated to set up. If, however, the major parts of the class hierarchies already have been established, the use of ***Factory Method*** allows for flexible variations of `Product` instantiations.

32.12. The Visitor design pattern

Lecture 8 - slide 37

The ***Visitor*** design pattern is typically connected to the ***Composite*** design pattern, which we have discussed in Section 32.1. Recall that a ***Composite*** gives rise to a tree of objects, all of which expose a uniform client interface. The ***Visitor*** design pattern is about a particular organization of the operations that visit each node in such a tree.

Relative to the ***Composite*** class diagram, shown in Figure 32.1, we will discuss two different organizations of the tree visiting operations:

- The natural object-oriented solution:
    - One method per operation per `Component` class
- The ***Visitor*** solution
    - All methods that pertain to a given operation are refactored and encapsulated in its own class

The natural object-oriented solution, mentioned first, is the solution that falls out of the ***Composite*** design pattern. We will illustrate it in the context of the `IntSequence` ***Composite*** in Section 32.13.

The ***Visitor*** solution is an alternative - and more complicated - organization which keeps all operations of a given traversal together. This is the solution of the ***Visitor*** design pattern. It will be exemplified in Section 32.15.

32.13. Natural object-oriented IntSequence traversals

Lecture 8 - slide 38

We have studied the integer sequence composite in appendix Section 58.3. The class diagram of this particular ***Composite*** is shown in Figure 58.1. Please recapitulate the essence of the integer sequence idea from there.

We will now discuss three different operations which need to visit each object in a integer sequence tree, such as the seven nodes of the tree shown in Figure 58.2. The operations are `Max`, `Min`, and `Sum`. `Min` and `Max` find the smallest/largest number in the tree respectively. `Sum` adds all numbers in the tree together.

Below we show the `Min`, `Max`, and `Sum` operations in four classes `IntSequence`, `IntSingular`, `IntInterval`, and `IntComposite`. All of the operation need to traverse the tree structure. Inner nodes in the composite tree are represented as instances of the class `IntCompSeq`, as shown in Program 32.26. The operations `Min`, `Max`, and `Sum` are implemented recursively in this class. A ***Composite*** is a recursive data structure which in a natural way calls for recursive processing. All this is archetypical for a composite structure.

```
public abstract class IntSequence {
  public abstract int Min {get;}
  public abstract int Max {get;}
  public abstract int Sum();
}
```

Program 32.23 *The abstract class IntSequence.*

```
public class IntInterval: IntSequence{

  private int from, to;

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

  public int From{
    get{return from;}
  }

  public int To{
    get{return to;}
  }

  public override int Min{
    get {return Math.Min(from,to);}
  }

  public override int Max{
    get {return Math.Max(from,to);}
  }

  public override int Sum(){
    int res = 0;
    int lower = Math.Min(from,to),
        upper = Math.Max(from,to);

    for (int i = lower; i <= upper; i++)
       res += i;
    return res;
  }
}
```

Program 32.24 *The class IntInterval.*

```
public class IntSingular: IntSequence{

  private int it;

  public IntSingular(int it){
    this.it = it;
  }

  public int TheInt{
     get{return it;}
  }

  public override int Min{
    get {return it;}
  }

  public override int Max{
    get {return it;}
  }

  public override int Sum(){
    return it;
  }
}
```

Program 32.25 *The class IntSingular.*

```
public class IntCompSeq: IntSequence{

  private IntSequence s1, s2;  // Binary sequence: Exactly two subsequences.

  public IntCompSeq(IntSequence s1, IntSequence s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  public IntSequence First{
    get{return s1;}
  }

  public IntSequence Second{
    get{return s2;}
  }

  public override int Min{
    get {return Math.Min(s1.Min, s2.Min);}
  }

  public override int Max{
    get {return Math.Max(s1.Max, s2.Max);}
  }

  public override int Sum(){
    return s1.Sum() + s2.Sum();
  }
}
```

Program 32.26 *The class IntCompSeq.*

In the web version of the material we show an integer sequence client which traverses a composite tree structure of integer sequences with use of the operations `Min`, `Max`, and `Sum`, see Program 32.27 (only on web). In Listing 32.28 (only on web) we also show the program output.

The programming of `Min`, `Max`, and `Sum` in the integer sequence classes, as shown above, is natural object-oriented programming of the traversals. Each of the four involved classes has a `Min`, `Max`, and a `Sum` operation. The operations are located in the immediate neighborhood of the data on which they rely. Everything is fine.

But the solution shown in this section is not a ***Visitor***. In the next section we will discuss and motivate the transition from the natural object-oriented solution to a visitor. After that we will reorganize `Min`, `Max` and `Sum` as visitors.

32.14. Towards a Visitor solution

Lecture 8 - slide 39

Before we study Visitors for integer sequence traversals we will discuss the transition from the natural object-oriented traversal to the ***Visitor*** design pattern.

The main idea of ***Visitor*** is to organize all methods that take part in a particular traversal, in a single `Visitor` class. In our example from Section 32.13 it means that we will have `MinVisitor`, `MaxVisitor`, and `SumVisitor` class. All of these classes share a common `Visitor` interface.

The following steps are involved the transition from natural object-oriented visiting to the ***Visitor*** design pattern:

- A `Visitor` interface and three concrete Visitor classes are defined
- The `Intsequence` classes are refactored - the traversal methods are moved to the visitor classes
- `Accept` methods are defined in the `IntSequence` classes. `Accept` takes a `Visitor` as parameter
- `Accept` passes `this` to the visitor, which in turn may activate `Accept` on components

From the web-version of the material we provide an SVG-animation that illustrates the transition.

Try the accompanying SVG animation

In the following section we will study an example. In the slipstream of the example we will explain and discuss additional details. The pros and cons of the ***Visitor*** solution are summarized in Section 32.16.

32.15. A Visitor example: IntSequence

Lecture 8 - slide 40

Let us now reorganize the integer sequence traversals from Section 32.13 to a ***Visitor***.

We provide three different traversals: find minimum, find maximum, and calculate sum. This will give rise to three different visitor objects: a minimum visitor, a maximum visitor, and a sum visitor of types `MinVisitor`, `MaxVisitor`, and `SumVisitor` respectively. The three classes implement a common `Visitor` interface. Each of the visitors will have `VisitIntInterval`, `VisitSingular`, and `VisitCompSeq` methods. As a naming issue, we chose to use the name `Visit` for all of them. This choice relies on method overloading. With these considerations we are able to understand the `Visitor` interface shown in Program 32.29.

```
public interface Visitor{
  int Visit (IntInterval iv);
  int Visit (IntSingular iv);
  int Visit (IntCompSeq iv);
}
```

Program 32.29 *The Visitor Interface.*

The abstract superclass in the integer sequence ***Composite*** design pattern, which we presented in Program 32.23, can now be reduced to a single method, which takes a `Visitor` object as parameter. The method is usually called `Accept`.

```
public abstract class IntSequence {
  public abstract int Accept(Visitor v);
}
```

Program 32.30 *The abstract class IntSequence.*

The idea behind the `Accept` method is to delegate the responsibility of a particular traversal to a given `Visitor` object. In the class `IntInterval`, shown below in Program 32.31, we see that `Accept` passes the current object (the `IntInterval` object) to the visitor. This is done in line 19. The same happens in `Accept` of `IntSingular` (line 14 of Program 32.32) and in `Accept` of `IntCompSeq` (line 19 of Program 32.33).

```
1  public class IntInterval: IntSequence{
2  
3    private int from, to;
4  
5    public IntInterval(int from, int to){
6      this.from = from;
7      this.to = to;
8    }
9  
10   public int From{
11     get{return from;}
12   }
13 
14   public int To{
15     get{return to;}
16   }
17 
18   public override int Accept(Visitor v){
19     return v.Visit(this);
20   }
21 }
```

Program 32.31 *The class IntInterval.*

```
1  public class IntSingular: IntSequence{
2  
3    private int it;
4  
5    public IntSingular(int it){
6      this.it = it;
7    }
8  
9    public int TheInt{
10      get{return it;}
11   }
12 
13   public override int Accept(Visitor v){
14     return v.Visit(this);
15   }
16 }
```

Program 32.32 *The class IntSingular.*

```
public class IntCompSeq: IntSequence{

  private IntSequence s1, s2;  // Binary sequence: Exactly two subsequences.

  public IntCompSeq(IntSequence s1, IntSequence s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  public IntSequence First{
    get{return s1;}
  }

  public IntSequence Second{
    get{return s2;}
  }

  public override int Accept(Visitor v){
    return v.Visit(this);
  }
}
```

Program 32.33 *The class IntCompSeq.*

It is now time to program the visitor classes (the classes that implement the `Visitor` interface of Program 32.29).

The `Visit` methods on intervals and singulars (the leafs in the composite tree) just extract information from the passed tree node. Thus, the `Visit` methods extract information from the objects that hold the essential information (this is the objects that provide the `Accept` methods). The `Visit` methods on the inner tree nodes (of type `IntCompSeq`) are more interesting. They call `Accept` methods on subtrees of the inner tree node. This is highlighted with blue color in Program 32.34, Program 32.35, and Program 32.36.

```
public class MinVisitor: Visitor{
  public int Visit (IntInterval iv){
    return Math.Min(iv.From, iv.To);
  }

  public int Visit (IntSingular iv){
    return iv.TheInt;
  }

  public int Visit (IntCompSeq iv){
    return Math.Min(iv.First.Accept(this),
                    iv.Second.Accept(this));
  }
}
```

Program 32.34 *The class MinVisitor.*

```
public class MaxVisitor: Visitor{
  public int Visit (IntInterval iv){
    return Math.Max(iv.From, iv.To);
  }

  public int Visit (IntSingular iv){
    return iv.TheInt;
  }

  public int Visit (IntCompSeq iv){
    return Math.Max(iv.First.Accept(this),
                    iv.Second.Accept(this));
  }
}
```

Program 32.35   *The class MaxVisitor.*

```
public class SumVisitor: Visitor{
  public int Visit (IntInterval iv){
    int res = 0;
    int lower = Math.Min(iv.From,iv.To),
        upper = Math.Max(iv.From,iv.To);

    for (int i = lower; i <= upper; i++)
       res += i;
    return res;
  }

  public int Visit (IntSingular iv){
    return iv.TheInt;
  }

  public int Visit (IntCompSeq iv){
    return (iv.First.Accept(this) +
            iv.Second.Accept(this));
  }
}
```

Program 32.36   *The class SumVisitor.*

As it appears, each `Accept` method in the ***Composite*** calls a `Visit` method in a `Visitor` class, which in turn may call one or more `Accept` methods on a composite constituents. This leads to *indirect recursion* in between `Accept` methods and `Visit` methods. Compared with the natural object-oriented solution, which uses *direct recursion*, this is more complicated to understand.

The indirect recursion, pointed out above, may also be understood as a simulation of *double dispatching*. First, we dispatch on the `Visitor` object and next we dispatch on an object from the composite tree structure. Most object-oriented programming language only allows *single dispatching* - corresponding to message passing via a virtual method. This can be generalized to *multiple dispatching*, where the dynamic type of several objects determine which method to activate. The object-oriented part of Common Lisp - CLOS [Keene89] - supports multiple dispatching.

In Program 32.37 we show a client program with an integer sequence composite structure (line 7-15), three visitors (line 16-18), and sample activations of tree traversals (highlighted in line 21, 22, and 28). The output of the program is shown in Listing 32.38 (only on web).

```
using System;

class SeqApp {

  public static void Main(){

    IntSequence isq =
      new IntCompSeq(
            new IntCompSeq(
              new IntInterval(3,5), new IntSingular(-7) ),
            new IntCompSeq(
              new IntInterval(12,7), new IntCompSeq(
                                         new IntInterval(18,-18),
                                         new IntInterval(3,5)
                                         )));
    Visitor min = new MinVisitor();
    Visitor max = new MaxVisitor();
    Visitor sum = new SumVisitor();


    Console.WriteLine("Min: {0} Max: {1}", isq.Accept(min),
                                           isq.Accept(max));

//  Alternative activation of Visit methods:
//  Console.WriteLine("Min: {0} Max: {1}", min.Visit((IntCompSeq)isq),
//                                         max.Visit((IntCompSeq)isq));

    Console.WriteLine("Sum: {0}", isq.Accept(sum));
  }
}
```

Program 32.37 *A sample application of IntSequences and visitors.*

32.16. Visitors - Pros and Cons

Lecture 8 - slide 41

As it is already clear from our explanation of ***Visitor*** in Section 32.15 there are both advantages and disadvantages of this design pattern.

We summarize the consequences of ***Visitor*** in the following items:

- A new kind of traversal can be added without affecting the classes of the ***Composite***
- A ***Visitor*** encapsulates all methods related to a particular traversal
- State related to a traversal can - in a natural way - be represented in the ***Visitor***
- If a new class is added to the ***Composite*** all ***Visitor*** classes are affected
- The indirect recursion that involves `Accept` and the `Visit` methods is more complex than the direct recursion in the natural object-oriented solution

Visitor is frequently used for processing of abstract syntax trees in compilation tools

In case you are going to study compilers implemented the object-oriented way, you will most likely encounter ***Visitors*** for such tasks as type checking and code generation.

32.17. References

[Keene89]	Sonya E. Keene, *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.
[Gamma96]	E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, 1996.
[Midi-sample]	The generated MIDI file http://www.cs.aau.dk/~normark/oop-csharp/midi/song.mid
[Mip-jan-08]	MIP Exam January 2008 (In Danish) http://www.cs.aau.dk/~normark/oop-07/html/mip-jan-08/opgave.html
[Song-and-timednote-classes]	The auxiliary classes TimedNote and Song http://www.cs.aau.dk/~normark/oop-07/html/mip-jan-08/c-sharp/mip.cs
[Factory-method]	Wikipedia: Design pattern: Factory Method http://en.wikipedia.org/wiki/Factory_method
[Abstract-factory]	Wikipedia: Design pattern: Abstract Factory http://en.wikipedia.org/wiki/Abstract factory

33. Fundamental Questions about Exception Handling

With this chapter we start the lecture about exception handling. We could as well just use the word "error handling". Before we approach object-oriented exception handling we will in this chapter discuss error handling broadly and from several perspectives. In Chapter 34 we will discuss non-OO, conventional exception handling. In Chapter 35 we encounter object-oriented exception handling. Finally, in Chapter 36 we discuss exception handling in C#. Chapter 36 is the main chapter in the lecture about exception handling.

33.1. What is the motivation?

Lecture 9 - slide 2

The following items summarize why we should care about error handling:

- *Understand* the nature of errors
  - "An error is not just an error"
- *Prevent* as many errors as possible in the final program
  - Automatically - via tools
  - Manually - in a distinguished testing effort
- Make programs more *robust*
  - A program should be able to resist and survive unexpected situations

33.2. What is an error?

Lecture 9 - slide 3

The word "error" is often used in an undifferentiated way. We will now distinguish between errors in the development process, errors in the source program, and errors in the executing program.

- Errors in the design/implementation *process*
  - Due to a wrong decision at an early point in time - a mental flaw
- Errors in the *source program*
  - Illegal use of the programming language
  - Erroneous implementation of an algorithm
- Errors in the *program execution* - run time errors
  - Exceptions - followed by potential handling of the exceptions

Errors in the development process **may** lead to errors in the source program.

Errors in the source program **may** lead to errors in the running program

33.3. What is normal? What is exceptional?

Lecture 9 - slide 4

I propose that we distinguish between "normal aspects" and "exceptional aspects" when we write a program. Without this distinction, many real-world programs will become unwieldy. The separation between normal aspects and exceptional aspects adds yet another dimension of structure to our programs.

In many applications and libraries, the programming of the normal aspects leads to nice and well-proportional solution. When exceptional aspects (error handling) are brought in, the normal program aspects are polluted with error handling code. In some situations the normal program aspects are totally dominated by exceptional program aspects. This is exemplified in Section 34.2.

Below we characterize the normal program aspects and the exceptional program aspects.

- Normal program aspects
  - Situations anticipated and dealt with in the conventional program flow
  - Programmed with use of selective and iterative control structures
- Exceptional program aspects
  - Situations anticipated, but not dealt with "in normal ways" by the programmer
    - Leads to an exception
    - Recoverable via exception handling. Or non-recoverable
  - Situations not anticipated by the programmer
    - Leads to an exception
    - Recoverable via exception handling. Or non-recoverable
  - Problems beyond the control of the program

Let us assume that we program the following simple factorial function. Recall that "*n factorial*" = `Factorial(`*n*`)` = *n*! = *n* * (*n* -1) * ... * 1.

```
public static long Factorial(int n){
  if (n == 0)
    return 1;
  else return n * Factorial(n - 1);
}
```

The following problems may appear when we run the `Factorial` function:

1. **Negative input** : If `n` is negative an infinite recursion will appear. It results in a `StackOverflowException` .
2. **Wrong type of input** : In principle we could pass a string or a boolean to the function. In reality, the compiler will prevent us from running such a program, however.
3. **Wrong type of multiplication** : The operator my be redefined or overloaded to non-multiplication.
4. **Numeric overflow of returned numbers** : The type `long` cannot contain the result of `20!` , but not `21!` .
5. **Memory problem** : We may run out of RAM memory during the computation.
6. **Loss of power** : The power may be interrupted during the computation.
7. **Machine failure** : The computer may fail during the computation.

8. **Sun failure** : The Sun may be extinguished during the computation.

Problem 1 should be dealt with as a normal program aspects. As mentioned, the problem in item 2 is prevented by the analysis of the compiler. Problem 3 is, in a similar way, prevented by the compiler. Problem 4 is classified as an anticipated exceptional aspect. Problem 4 could, alternatively, be dealt with by use of another type than *long*, such a `BigInteger` which allows us to work with arbitrary large integers. (`BigInteger` is not part of the .Net 3.5 libraries, however). Problem 5 could also be foreseen as an anticipated exception. Problem 5, 7, and 8 are beyond the control of the program. In extremely critical applications it may, however, be considered to deal with (handle) problem 6 and 7.

With use of normal control structures, a different (although a hypothetical) type `BigInteger`, and an iterative instead of a recursive algorithm we may rewrite the program to the following version:

```
public static BigInteger Factorial(int n){
  if (n >= 0){
    BigInteger res = 1;
    for(int i = 1; i <= n; i++)
      res = res * i;
    return res;
  }
  else throw new ArgumentException("n must be non-negative");
}
```

With this rewrite we have dealt with problem 1, 4, and 5. As an attractive alternative to the `if-else`, problem 1 could be dealt with by the precondition `n >= 0` of the `Factorial` method, see Section 50.1.

As it appears, we wish to distinguish between normal program aspects and exceptional program aspects via the programming language mechanisms used to deal with them. In C# and similar object-oriented languages, we have special means of expressions to deal with exceptions. The `Factorial` function shown above throws an exception in case of negative input. See Section 36.2 for details.

Above, we distinguish between different degrees of exceptional aspects. As a programmer, you are probably aware of something that can go wrong in your program. Other errors come as surprises. Some error situations, both the expected and the surprising ones, should be dealt with such that the program execution survives. Others will lead to program termination. Controlled program termination, which allows for smooth program restart, will be an important theme in this lecture.

33.4. When are errors detected?

Lecture 9 - slide 6

It is attractive to find errors as early as possible

Our next question cares about the point in time where you - the program developer - realize the problem. It should be obvious that we wish to identify troubles as soon as possible.

We identify the following error identification times.

- During design and programming - *Go for it.*
- During compilation - syntax errors or type errors - *Attractive.*
- During testing - *A lot of hard work. But necessary.*
- During execution and final use of the program
    - Handled errors - *OK. But difficult.*
    - Unhandled errors - *A lot of frustration.*

If we are clever enough, we will design and program our software such that errors do not occur at all. However, all experience shows that this is not an easy endeavor. Still, it is good wisdom to care about errors and exception handling early in the development process. Problems that can be dealt with effectively at an early point in time will save a lot of time and frustrations in the latter phases of the development process.

Static analysis of the program source files, as done by the front-end of the compiler, is important and effective for relatively early detection of errors. The more errors that can be detected by the compiler before program execution, the better. Handling of errors caught by the compiler requires very little work from the programmers. This is at least the case if we compare it with testing efforts, described next.

Systematic test deals with sample execution of carefully prepared program fragments. The purpose of testing is to identify errors (see also Section 54.1). Testing activities are very time consuming, but all experience indicates that it is necessary. We devote a lecture, covered by Chapter 56 in this material, to testing. We will in particular focus on unit test of object-oriented programs.

Software test is also known as validation in relation to the specification of the software. Alternatively, the program may be formally verified up against a specification. This goes in the direction of a *mathematical proof*, and an area known as *model-checking*.

Finally, some errors may creep through to the end-use of the program. Some of these errors could and should perhaps have been dealt with at an earlier point in time. But there will remain some errors in this category. Some can be handled and therefore hidden behind the scene. A fair amount cannot be handled. Most of the discussion in this and the following three chapters are about (handled and unhandled) errors that show up in the program at execution time.

33.5. How are errors handled?

Lecture 9 - slide 7

Assuming that we now know about the nature of errors and when they appear in the running program, it is interesting to discuss what to do about them. Here follows some possibilities.

- **Ignore**
    - False alarm - Naive
- **Report**
    - Write a message on the screen or in a log - Helpful for subsequent correction of the source program
- **Terminate**
    - Stop the program execution in a controlled an gentle way - Save data, close connections
- **Repair**
    - Recover from the error in the running program - Continue normal program execution when the problem is solved

The first option - false alarm - is of course naive and unacceptable from a professional point of view. It is naive in the sense that shortly after we have ignored the error another error will most certainly occur. And what should then be done?

The next option is to tell the end-user about the error. This is naive, almost in the same way as false alarm. But the reporting option is a very common reaction from the programmer: "*If something goes wrong, just print a message on standard output, and hopefully the problem will vanish.*" At least, the user will be aware that something inappropriate has happened.

The termination option is often the most viable approach, typically in combination with proper reporting. The philosophy behind this approach is that errors should be corrected when they appear. The sooner the better. The program termination should be controlled and gentle, such that it is possible to continue work when the problem has been solved. Data should be saved, and connections should be closed. It is bad enough that a program fails "today". It is even worse if it is impossible start the program "tomorrow" because of corrupted data.

Repair and recovery at run-time is the ultimate approach. We all wish to use robust and stable software. Unfortunately, there are some problems that are very difficult to deal with by the running program. To mention a few, just think of broken network connections, full harddisks, and power failures. It is only in the most critical applications (medical, atomic energy, etc) that such severe problems are dealt with explicitly in the software that we construct. Needless to say, it is very costly to built software that takes such problems into account.

33.6. Where are errors handled?

Lecture 9 - slide 8

The last fundamental question is about the place in the program where to handle errors. Should we go for local error handling, or for handling at a more remote place in the program.

- Handle errors at the place in the program where they occur
    - If possible, this is the easiest approach
    - Not always possible nor appropriate
- Handle errors at another place
    - Along the calling chain
    - Separation of concerns

If many errors are handled in the immediate proximity of the source of the error, chances are that a small and understandable program becomes large, unwieldy, and difficult understand. *Separation of concerns* is worth considering. One concern is the normal program aspects (see Section 33.3). Another concern is exception handling. The two concerns may be dealt with in different corners or the program. Propagation of errors from one place in a C# program to another will be discussed in Section 36.7 of this material.

34. Conventional Exception Handling

Before we approach exception handling in object-oriented programs we will briefly take a look at some conventional ways to deal with errors. You can, for instance, think of these as error handling techniques in C programming.

34.1. Exception Handling Approaches

Lecture 9 - slide 10

One way to deal with errors is to bring the error condition to the attention of the user of the program. (See Section 33.5). Obviously, this is done in the hope that the user has a chance to react on the information he or she receives. Not all users can do so.

If error messages are printed to streams (files) in conventional, text based user interfaces, it is typical to direct the information to the *standard error* stream instead of the *standard output* stream.

- Printing error messages
    - `Console.Out.WriteLine(...)` *or* `Console.Error.WriteLine(...)`
    - Error messages on standard output are - in general - a bad idea

We identify the following conventional exception handling approaches:

- Returning error codes
    - Like in many C programs
    - In conflict with a functional programming style, where we need to return data
- Set global error status variables
    - Almost never attractive
- Raise and handle exceptions
    - A special language mechanism to raise an error
    - Rules for propagation of errors
    - Special language mechanisms for handling of errors

When a function is used in imperative C programming, the value returned by the function can be used to signal an error condition to its caller. Many functions from the standard C library signal errors via the value returned from the function. Unfortunately, varying conventions are applied. In some functions, such as `main`, a non-zero value communicates a problem to the operating environment. In functions that return pointers (such as `malloc`) a `NULL` value typically indicates an error condition. In other functions, such as `fgetc` and `fputc`, the distinguished `EOF` (end of file) value is used as an error value. Other functions, such as `mktime`, use -1 as an error value.

As an supplementing means, a global variable can be used for signaling more details about an error. In C programming, the variable `errno` from the standard library `errno.h` is often used. When a function returns an error value (as discussed above), the value of `errno` is set to a value that gives more details about the error. Some systems use `errno` as an index to an array of error messages.

Use of error return values and global error variables is not a good solution to the error handling problem. Therefore, most contemporary languages come with more sophisticated facilities for raising, propagating and handling exceptions. The C# error handling facilities are covered specifically in Section 36.2, Section 36.3, and Section 36.7.

34.2. Mixing normal and exceptional cases

Lecture 9 - slide 11

Before we enter the area of object-oriented exception handling, and exception handling in C#, we will illustrate the danger of mixing "normal program aspects" and "exceptional program aspects". See also the discussion in Section 33.3.

In Program 34.1 a small program, which copies a file into another file, is organized in the `Main` method in a C# program. The string array passed to `Main` is supposed to hold the names of the source and target files. Most readers will probably agree that the program fragment shown in Program 34.1 is relatively clear and straightforward.

```
using System;
using System.IO;

public class CopyApp {

  public static void Main(string[] args) {
    FileInfo   inFile  = new FileInfo(args[0]),
               outFile = new FileInfo(args[1]);
    FileStream inStr   = inFile.OpenRead(),
               outStr  = outFile.OpenWrite();
    int c;
   do{
      c = inStr.ReadByte();
      if(c != -1) outStr.WriteByte((byte)c);
   } while (c != -1);

   inStr.Close();
   outStr.Close();
 }
}
```

Program 34.1   *A file copy program.*

We will now care about possible issues that can go wrong in our file copy program. The result can be seen in Program 34.2, where all **red** aspects are oriented towards error handling. Some of the error handling issues are quite realistic. Others may be slightly exaggerated with the purpose of making our points.

```
1  using System;
2  using System.IO;
3  
4  public class CopyApp {
5  
6    public static void Main(string[] args) {
7      FileInfo inFile;
8      do {
9          inFile = new FileInfo(args[0]);
10         if (!inFile.Exists)
11           args[0] = "some other input file name";
12     } while (!inFile.Exists);
13 
14     FileInfo outFile;
15     do {
16         outFile = new FileInfo(args[1]);
17         if (outFile.Exists)
18           args[1] = "some other output file name";
19     } while (outFile.Exists);
20 
21     FileStream inStr   = inFile.OpenRead(),
22                outStr  = outFile.OpenWrite();
23     int c;
24     do{
25        c = inStr.ReadByte();
26        if(c != -1) outStr.WriteByte((byte)c);
27        if (StreamFull(outStr))
28          DreamCommand("Fix some extra room on the disk");
29     } while (c != -1);
30 
31     inStr.Close();
32     if (!FileClosed(inStr))
33       DreamCommand("Deal with input file which cannot be closed");
34 
35     outStr.Close();
36     if (!FileClosed(outStr))
37       DreamCommand("Deal with output file which cannot be closed");
38   }
39 }
```

Program 34.2 *A file copy program with excessive error handling.*

The line intervals 8-12 and 15-19 deal with non-existing/existing input/output files respectively. It is a problem if the input file is non-existing, and it may be problematic to overwrite an existing output file. Line 27-28 deals with memory problems, in case there is not enough room for the output file. The line intervals 32-33 and 36-37 address file closing problems. The methods `DreamCommand`, `FileClosed`, and `StreamFull` are imaginary abstractions, which are needed to complete and compile the version of the file copy program shown in Program 34.2.

The important lesson to learn from the example above is that the original "normal file copying aspects" in Program 34.1 almost disappears in between the error handling aspects of Program 34.2.

35. Object-oriented Exception Handling

It may be asked if there is a solid link between object-oriented programming and exception handling. I see two solid object-oriented contributions to error handling. The contributions are (1) representation of an error as an object, and (2) classification of errors in class inheritance hierarchies. These contributions will be explained at an overall level in this chapter. In Chapter 36 we will address the same issues relative to C#.

35.1. Errors as Objects

Lecture 9 - slide 13

An error is characterized by several pieces of information. It is attractive to keep these informations together. By keeping relevant error information together it becomes easier to propagate error information from one place in a program to another.

Seen in this light, it is obvious that an error should be represented as an object.

All relevant knowledge about an error is encapsulated in an object

- Encapsulation of relevant error knowledge
  - Place of occurrence (class, method, line number)
  - Kind of error
  - Error message formulation
  - Call stack information
- Transportation of the error
  - From the place of origin to the place of handling
  - Via a special **`throw`** mechanism in the language

An error is an object. Objects are instances of classes. Therefore there will exist classes that describe common properties of errors. In the next section we will discuss the organization of these classes in a class hierarchy.

35.2. Classification of Errors

Lecture 9 - slide 14

There are many kinds of errors: Fatal errors, non-fatal errors, system errors, application errors, arithmetic errors, IO errors, software errors, hardware errors, etc. It would be very helpful if we could bring order into this mess.

In Section 35.1 we realized that a concrete error can be represented as an instance of a class, and consequently that we can deal with *types of errors*. Like other types, different types of errors can therefore be organized in type hierarchies. At the programming language level we can define a set of error/classes, and we can organize these in an inheritance hierarchy.

Errors can be classified and specialized by means of a class hierarchy

Figure 35.1 shows an outline of type hierarchy for different kinds of errors. The concrete C# counterpart to this figure is discussed in Section 36.4.

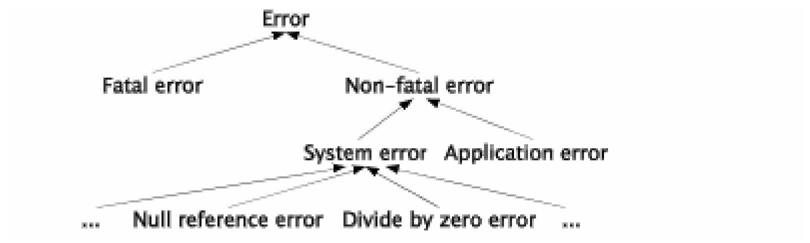

Figure 35.1 *A sample error classification hierarchy*

As hinted by the introductory words of this section, there may be several different classifications of errors. The classification in Figure 35.1 only represents one such possibility. If multiple inheritance is available (see Section 27.4 and Section 27.5) multiple error classification schemes may coexist.

36. Exceptions and Exception Handling in C#

Chapter 33, Chapter 34, and Chapter 35 have provided a context for this chapter. Warmed up in this way, we will now discuss different aspects of exceptions and exception handling in C#.

36.1. Exceptions in a C# program

Lecture 9 - slide 16

Let us start with an simple example. Several variants of the example will appear throughout this chapter. In Program 36.1 the `int` table, declared and instantiated in line 6, is accessed by the expression `M(table, idx)` in line 9. In `M` we happen to address a non-existing element in the table. Recall that an array of 6 elements holds the elements `table[0]`.. . `table[5]`. Therefore the cell addressed by `table[6]` is non-existing. Consequently, the execution of `M(table, idx)` in line 9 causes an error (index out of range). This is a *run-time error*, because the error happens when the program is executing. You can see this as a contrast to *compile-time errors*, which are identified before the program starts. In C#, a run-time error is materialized as an exception, which is an instance of class `Exception` or one of its subclasses. After its creation the exception is *thrown*. Throwing an exception means that the exception object is brought to the attention of exception handlers (catchers, see Section 36.3) which have a chance to react on the error condition represented by the exception. In the example shown in Program 36.1 the thrown exception is not handled.

```
1  using System;
2
3  class ExceptionDemo{
4
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8
9      M(table, idx);
10   }
11
12   public static void M(int[] table, int idx){
13     Console.WriteLine("Accessing element {0}: {1}",
14                        idx, table[idx]);
15   }
16
17 }
```

Program 36.1 *A C# program with an exception.*

The output of Program 36.1 is shown in Listing 36.2. The effect of the `WriteLine` command in line 13 never occurs, because the error happens before `WriteLine` takes effect. The output in Listing 36.2 is therefore produced exclusively by the unhandled exception. We can see that the exception is classified as an `IndexOutOfRangeException`, which is quite reasonable. We can also see the stack trace from the beginning of the program to the place where the exception is thrown: `Main`, `M` (read from bottom to top in Listing 36.2).

```
Unhandled Exception: System.IndexOutOfRangeException:
 Index was outside the bounds of the array.
     at ExceptionDemo.M(Int32[] table, Int32 idx)
     at ExceptionDemo.Main()
```

Listing 36.2 *Output from the C# program with an exception.*

36.2. The try-catch statement C#

Lecture 9 - slide 17

In Section 36.1 we illustrated an unhandled exception. The error occurred, the exception object was formed, it was propagated through the calling chain, but it was never reacted upon (handled).

We will now introduce a new control structure, which allows us to handle exceptions, as materialized by objects of type `Exception`. Handling an exception imply in some situations that we attempt to recover from the error which is represented by the exception, such that the program execution can continue. In other situations the handling of the exception only involves a few repairs or state changes just before the program terminates. This is typically done to save data or to close connections such that the program can start again when the error in the source program has been corrected.

The try-catch statement allows us handle certain exceptions instead of stopping the program

The syntax of the new control structure is as shown below.

```
try
  try-block
catch (exception-type-1 name)
  catch-block-1
catch (exception-type-2 name)
  catch-block-2
...
```

Syntax 36.1 *The syntax of try-catch statement C#*

`try-block` and `catch-block-`*i* are indeed block statements. It means that braces `{...}` are mandatory after **try** and after **catch**. Even if only a single action happens in **try** or **catch**, the action must be dressed as a block.

Let us assume that we are interested in handling the exceptions that are caused by execution of some command *c*. This can be arranged by embedding *c* in a try-catch control structure. It can also be arranged if another command *d*, which directly or indirectly activates *c*, is embedded in a try-catch control structure.

If an exception of a given type occurs, it can be handled in one of the matching catch clauses. A catch clause matches an exception object *e*, if the type of *e* is a subtype of `exception-type-i` (as given in one of the catch clauses). The matching of exceptions and catch clauses are attempted in the order provided by the catch clauses in the try control structure. Notice that each catch clause in addition specifies a name, to which the given exception object will be bound (in the scope of the handler). The names in catch clauses are similar to formal parameter names in methods.

Syntax 36.1 does not reflect the whole truth. The names of exceptions, next to the exception types, may be missing. It is even possible to have a catch clause without specification of an exception type. There is also an optional **finally** clause, which we will discuss in Section 36.9.

36.3. Handling exceptions in C#

Lecture 9 - slide 18

Now that we have introduced the try-catch control structure let us handle the exception in Program 36.1. In Program 36.3 - inside `M` - around the two activations of `WriteLine`, we introduce a try-catch construct. If an `IndexOutOfRangeException` occurs in the try part, the control will be transferred to the neighbor catch part in which we adjust the index (using the method `AdjustIndex`), and we print the result. The program now prints "`We get element number 5: 15`".

Notice that once we have left the try part, due to an error, we will not come back to the try part again, even if we have repaired the problem. In Program 36.3 this means that the `Console.WriteLine` call in 16-17 is never executed. After having executed the catch clause in line 19-23, line 24 is executed next, and `M` returns to `Main`.

```
1  using System;
2  
3  class ExceptionDemo{
4  
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8  
9      M(table, idx);
10   }
11 
12   public static void M(int[] table, int idx){
13     try{
14       Console.WriteLine("Accessing element {0}: {1}",
15                          idx, table[idx]);
16       Console.WriteLine("Accessing element {0}: {1}",
17                          idx-1, table[idx-1]);
18     }
19     catch (IndexOutOfRangeException e){
20       int newIdx = AdjustIndex(idx,0,5);
21       Console.WriteLine("We get element number {0}: {1}",
22                          newIdx, table[newIdx]);
23     }
24     Console.WriteLine("End of M");
25   }
26 
27 
28 
29   public static int AdjustIndex(int i, int low, int high){
30     int res;
31     if (i < low)
32       res = low;
33     else if (i > high)
34       res = high;
35     else res = i;
36 
37     return res;
38   }
39 }
```

Program 36.3 *A C# program with a handled exception.*

In the example above we handled the exception in the immediate neighborhood of the offending statement. It is also possible to handle the exception at a more remote place in the program, but always along the path of activated, non-completed methods. We will illustrate this in Section 36.7.

Exercise 9.1. *Exceptions in Convert.ToDouble*

The static methods in the static class `System.Convert` are able to convert values of one type to values of another type.

Consult the documentation of `System.Convert.ToDouble`. There are several overloads of this method. Which exceptions can occur by converting a string to a double?

Write a program which triggers these exceptions.

Finally, supply handlers of the exceptions. The handlers should report the problem on standard output, rethrow the exception, and then continue.

36.4. The hierarchy of exceptions in C#

Lecture 9 - slide 19

In this section we will take a concrete look at the classification of exceptions in C#. Our general discussion of this topic can be found in Section 35.2.

The following shows an excerpt the `Exception` class tree in C#. The tree is shown by textual indentation. Thus, the classes `ApplicationException` and `SystemException` are sons (and subclasses) of `Exception`.

- Exception
    - ApplicationException
        - *Your own exception types*
    - SystemException
        - ArgumentException
            - ArgumentNullException
            - ArgumentOutOfRangeException
        - DivideByZeroException
        - IndexOutOfRangeException
        - NullReferenceException
        - RankException
        - StackOverflowException
        - IOException
            - EndOfStreamException
            - FileNotFoundException
            - FileLoadException

Notice first that the `Exception` class tree is not the whole story. There are many more exception classes in the C# libraries than shown above.

Exceptions of type `SystemException` are thrown by the common language runtime (the virtual machine) if some error condition occurs. System exceptions are nonfatal and recoverable. As a programmer, you are also welcome to throw a `SystemException` object (or more precisely, an object of one of the subclasses of `SystemException`) from a program, which you are writing.

An `ArgumentException` can be thrown if a an operation receives an illegal argument. The programmer of the operation decides which arguments are legal and illegal. The two shown subclasses of `ArgumentException` reflect that the argument cannot be `null` and that the argument is outside its legal range respectively.

The `DivideByZeroException` occurs if zero is used as a divisor in a division. The `IndexOutOfRangeException` occurs if an an array is accessed with an index, which is outside the legal bounds. The `NullReferenceExceptions` occurs in an expression like `ref.name` where ref is `null` instead of a reference to an object. The `RankException` occurs if an array with the wrong number of dimensions is passed to an operation. The `StackOverflowException` occurs if the memory space devoted to non-completed method calls is exhausted. The `IOException` (in the namespace `System.IO`) reflects different kinds of errors related to file input and file output.

`ApplicationExceptions` are "thrown when a non-fatal application error occurs" (quote from MSDN). The common runtime system throws instances of `SystemException`, not `ApplicationException`. Originally, the exception classes that you program in your own code were intended to be subclasses of `ApplicationException`. In version 3.5 of the .NET framework, Microsoft recommends that your own exceptions are programmed as subclasses of `Exception` [exceptions-best-practices].

You are encouraged to identify and throw exceptions which are specializations of `SystemException`. By (re)using existing exception types, it becomes possible for the system, or for third-party program contributions, to catch the exceptions that you throw from your own code.

Exercise 9.2. *Exceptions in class Stack*

In the lecture about inheritance we specialized the abstract class `Stack`.

Now introduce exception handling in your non-abstract specialization of `Stack`. I suggest that you refine your own solution to the previous exercise. It is also possible to refine my solution.

More specifically, introduce one or more stack-related exception classes. The slide page "Raising and throwing exceptions in C#" tells you how to do it. Make sure to specialize the appropriate pre-existing exception class!

Arrange that `Push` on a full stack and that `Pop`/`Top` on an empty stack throw one of the new exceptions. Also, in the abstract stack class, make sure that `ToggleTop` throws an exception if the stack is empty, or if the stack only contains a single element.

Finally, in a sample client program such as this one, handle the exceptions that are thrown. In this exercises it is sufficient to report the errors on standard output.

Exercise 9.3. *More exceptions in class Stack*

In continuation of the previous exercise, we now wish to introduce the following somewhat unconventional handling of stack exceptions:

- If you push an element on a full stack throw half of the elements away, and carry out the pushing.
- If you pop/top an empty stack, push three dummy elements on the stack, and do the pop/top operation after this.

With these ideas, most stack programs will be able to terminate normally (run to the end).

I suggest that you introduce yet another specialization of the stack class, which specializes `Push`, `Pop`, and `Top`. The specialized stack operations should handle relevant stack-related exceptions, and delegate the real work to its superclass. Thus, in the specialized stack class, each stack operation, such as `Push`, you should embed `base.push(el)` in a **try-catch** control structure, which repairs the stack - as suggested above - in the catch clause.

36.5. The class System.Exception in C#

Lecture 9 - slide 20

The class `Exception` is the common superclass of all exception classes, and therefore it holds all common data and operations of exceptions. In this section we will examine the class `Exception` in some details.

- Constructors
  - Parameterless: **`Exception()`**
  - With an explanation: **`Exception(string)`**
  - With an explanation and an inner exception: **`Exception(string,Exception)`**
- Properties
  - `Message`: A description of the problem (string)
  - `StackTrace`: The call chain from the point of throwing to the point of catching
  - `InnerException`: The exception that caused the current exception
  - `Data`: A dictionary of key/value pairs.
    - For communication in between functions along the exception propagation chain.
  - *Others...*

`Exception` is a class (and instances of class `Exception` represents a concrete error). Therefore there exists constructors of class `Exceptions`, which (as usual) are used for initialization of a newly allocated `Exception` object. (See Section 12.4 for a general discussion of constructors).

The most useful constructor in class `Exception` takes a string, which holds an intuitive explanation of the problem. This string will appear on the screen, if a thrown exception remains unhandled. The third constructor, of the form `Exception(string,Exception)`, involves an inner exception. Inner exceptions will be discussed in Section 36.11.

As outlined above, an exception has an interface of properties (see Chapter 18) that give access to the data, which are encapsulated by the `Exception` object. You can access the message (originally provided as input to the constructor), the stack trace, a possible inner exception, and data in terms of a key-value dictionary (used to keep track of additional data that needs to travel together with the exception). For general information about dictionaries, see Chapter 46.

36.6. Handling more than one type of exception in C#

Lecture 9 - slide 21

We now continue the example from Section 36.3. In this section we will see how to handle multiple types of exceptions in a single try-catch statement.

The scene of Program 36.4 is similar to the scene in Program 36.3. In the catch clauses of the try-catch control structure we handle `NullReferenceException` and `DivideByZeroException`. On purpose, we do not yet handle `IndexOutOfRangeException`. Just wait a moment...

```
using System;

class ExceptionDemo{

  public static void Main(){
    int[] table = new int[6]{10,11,12,13,14,15};
    int idx = 6;

    M(table, idx);
  }

  public static void M(int[] table, int idx){
    try{
      Console.WriteLine("Accessing element {0}: {1}",
                         idx, table[idx]);
    }
    catch (NullReferenceException){
      Console.WriteLine("A null reference exception");
      throw;      // rethrowing the exception
    }
    catch (DivideByZeroException){
      Console.WriteLine("Divide by zero");
      throw;      // rethrowing the exception
    }

  }
}
```

Program 36.4 *A C# program with an exception handling attempt - not a success.*

When we run the program in Program 36.4 the two handlers do not match the exception that occurs (the `IndexOutOfRangeException` exception). Therefore the exception remains unhandled, and the program stops with the output shown in Listing 36.5.

Notice that we do not provide names of the exceptions in the catch clauses in Program 36.4. We could do so. But because the names are not used they would cause the compiler to issue warnings.

While we are here, let us dwell on the two catch clauses that actually appear in the try-catch statement in Program 36.4. The null reference exception describes the problem of executing `r.f` in state where `r` refers to `null`. The divide by zero exception describes the problem of executing `a/b` in state where `b` is zero. The catch clauses report the problems, but they do not handle them. Instead, both catch clauses rethrow the exceptions. This is done by `throw` in line 19 and 23. By rethrowing the exceptions, an outer exception handler (surrounding the try catch) or exception handlers along the dynamic calling chain will have a chance to make a repair. Reporting the exception is a typical temptation of the programmer. But the reporting itself does not solve the problem! Therefore you should rethrow the exception in order to let another part of the program have to chance to make an effective repair. Rethrowing of exceptions is discussed in Section 36.10.

```
Unhandled Exception:
 System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.5 *Output from the C# program with an unhandled exception.*

It was really too bad that we did not hit the `IndexOutOfRangeException` exception in Program 36.4. In Program 36.6 we will make a better job.

We extend the catch clauses with two new entries. We add the `IndexOutOfRangeException` and we add the root exception class `Exception`. Notice that the more general exception classes should always occur at the rear end of the list of catch clauses. The reason is that the catch clauses are consulted in the order they appear. (If the `Exception` catcher was the first one, none of the other would ever have a chance to take effect).

In the concrete example, the `IndexOutOfRangeException` clause is the first that matches the thrown exception. (Notice that newly added `Exception` clause also matches, but we never get that far). Therefore we get the output shown in Listing 36.7.

```
using System;

class ExceptionDemo{

  public static void Main(){
    int[] table = new int[6]{10,11,12,13,14,15};
    int idx = 6;

    M(table, idx);
  }

  public static void M(int[] table, int idx){
    try{
      Console.WriteLine("Accessing element {0}: {1}",
                        idx, table[idx]);
    }
    catch (NullReferenceException){
      Console.WriteLine("A null reference exception");
      throw;      // rethrowing the exception
    }
    catch (DivideByZeroException){
      Console.WriteLine("Divide by zero");
      throw;      // rethrowing the exception
    }
    catch (IndexOutOfRangeException){
      int newIdx = AdjustIndex(idx,0,5);
      Console.WriteLine("We get element number {0}: {1}",
                           newIdx, table[newIdx]);
    }
    catch (Exception){
      Console.WriteLine("We cannot deal with the problem");
      throw;     // rethrowing the exception
    }

  }

}
```

Program 36.6 *A C# program with an exception handling attempt - now successful.*

```
We get element number 5: 15
```

Listing 36.7 *Output from the C# program with a handled exception.*

Handle specialized exceptions before general exceptions

36.7. Propagation of exceptions in C#

Lecture 9 - slide 22

In the examples shown until now (see Program 36.3, Program 36.4, and Program 36.6) we have handled exceptions close to the place where they are thrown. This is not necessary. We can propagate an exception object to another part of the program, along the chain of the incomplete method activations.

In Program 36.8 there is a try-catch statement in `M` and (as a new thing) also in `Main`. The local catchers in `M` do not handle the actual exception (which still is of type index out of range). The handlers in `Main` do! When the error occurs in `M`, the local exception handlers all have a chance of handling it. They do not! Therefore the exception is *propagated* to the caller, which is `Main`. Due to the propagation of the exception, line 34 of `M` is never executed. The catchers around the activation of `M` in `Main` have a relevant clause that deals with `IndexOutOfRangeException`. It handles the problem by use of the static method `AdjustIndex`. After having executed the catch clause in `Main`, the command after **`try-catch`** in `Main` is executed (line 17). The output of Program 36.8 is shown in Listing 36.9.

```
1  using System;
2  
3  class ExceptionDemo{
4  
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8  
9      try{
10       M(table, idx);
11     }
12     catch (IndexOutOfRangeException){
13       int newIdx = AdjustIndex(idx,0,5);
14       Console.WriteLine("We get element number {0}: {1}",
15                            newIdx, table[newIdx]);
16     }
17     Console.WriteLine("End of Main");
```

```
  }

  public static void M(int[] table, int idx){
    try{
      Console.WriteLine("Accessing element {0}: {1}",
                         idx, table[idx]);
    }
    catch (NullReferenceException){
      Console.WriteLine("A null reference exception");
      throw;      // rethrowing the exception
    }
    catch (DivideByZeroException){
      Console.WriteLine("Dividing by zero");
      throw;     // rethrowing the exception
    }

    Console.WriteLine("End of M");
  }

}
```

Program 36.8 *A C# program with simple propagation of exception handling.*

```
We get element number 5: 15
End of Main
```

Listing 36.9 *Output from the C# program simple propagation.*

In order to illustrate a longer error propagation chain, we now in Program 36.10 introduce the calling chain

$$\texttt{Main} \longrightarrow \texttt{M} \longrightarrow \texttt{N} \longrightarrow \texttt{P}$$

```
using System;

class ExceptionDemo{

  public static void Main(){
    int[] table = new int[6]{10,11,12,13,14,15};
    int idx = 6;

    Console.WriteLine("Main");
    try{
      M(table, idx);
    }
    catch (IndexOutOfRangeException){
      M(table, AdjustIndex(idx,0,5));
    }
  }

  public static void M(int[] table, int idx){
    Console.WriteLine("M(table,{0})", idx);
    N(table,idx);
  }

  public static void N(int[] table, int idx){
    Console.WriteLine("N(table,{0})", idx);
    P(table,idx);
  }

  public static void P(int[] table, int idx){
    Console.WriteLine("P(table,{0})", idx);
    Console.WriteLine("Accessing element {0}: {1}",
                       idx, table[idx]);
  }

}
```

Program 36.10 *A C# program with deeper exception propagation chain.*

The error occurs in `P`, and it is handled in `Main`. Here is what happens when the expression `M(table, idx)` in line 11 is executed:

1. The method `M` calls method `N` , `N` calls `P` , and in `P` an exception is thrown.
2. The error is propagated back from `P` to `Main` via `N` and `M` , because there are no (relevant) handlers in `P` , `N` or `M` .
3. The exception is handled in `Main` , and as part of the handling `M` is called again: `M(table, AdjustIndex(idx,0,5))` .
4. As above, `M` calls `N` , `N` calls `P` , and `P` calls `WriteLine` . Now no errors occur.

Due to the tracing calls of `WriteLine` in `Main`, `M`, `N`, and `P` the output shown in Listing 36.11, in part, confirms the story told about. To obtain the full confirmation, consult Exercise 9.4.

```
Main
M(table,6)
N(table,6)
P(table,6)
M(table,5)
N(table,5)
P(table,5)
Accessing element 5: 15
```

Listing 36.11 *Output from the C# program deeper exception propagation.*

Notice the *yo-yo effect* caused by the error deep in the calling chain.

Exercise 9.4. *Revealing the propagation of exceptions*

We have written a program that reveals how exceptions are propagated. In the program output, we see that the calling chain is `Main`, `M`, `N`, `P`.

The program output does not, however, reveal that the chain is followed in reverse order in an attempt to find an appropriate exception handler.

Revise the program with handlers in `M`, `N`, and `P` that touch the exception without actually handling it. The handlers should reveal, on standard output, that `P`, `N`, and `M` are passed in an attempt to locate a relevant exception handler. Rethrow the exception in each case.

36.8. Raising and throwing exceptions in C#

Lecture 9 - slide 23

The `IndexOutOfRangeException`, which we have worked with in the previous sections, was raised by the system, as part of an illegal array indexing. We will now show how to explicitly raise an exception in our own program. We will also see how to define our own subclass of class `ApplicationException`.

The syntax of throw appears in in Syntax 36.2 and a simple example is shown next in Program 36.12. Notice the athletic metaphor behind throwing and catching.

```
throw exception-object
```

Syntax 36.2 *The syntax of exception throwing in C#*

```
...
throw new MyException("Description of problem");
...
```

Program 36.12 *A throw statement in C#.*

It is simple to define the class `MyException` as subclass of `ApplicationException`, which in turn is a subclass of `Exception`, see Section 36.4. Notice the convention that our own exception classes are subclasses of `ApplicationException`.

```
class MyException: ApplicationException{
  public MyException(String problem):
    base(problem){
  }
}
```

Program 36.13 *Definition of the exception class.*

It is recommended to adhere to a coding style where the suffixes (endings) of exception class names are "`...Exception`".

36.9. Try-catch with a finally clause

Lecture 9 - slide 24

A try-catch control structure can be ended with an optional **finally** clause. Thus, we really deal with a try-catch-finally control structure. In this section we will study the **finally** clause.

The syntax of try-catch-finally, shown in Syntax 36.3, is a natural extension of the try-catch control structure illustrated in Syntax 36.1. `try-block`, `catch-block`, and `finally-block` are all block statements. As explained in Section 36.2 is means that braces `{...}` are mandatory after **try**, **catch**, and **finally**.

```
try
  try-block
catch (exception-type name)
  catch-block
...
finally
  finally-block
```

Syntax 36.3   *The syntax of the try-catch-finally statement C#*

At least one **catch** or **finally** clause must appear in a **try** statement. The **finally** clause will be executed in all cases, both in case of errors, in case of error-free execution of try part, and in cases where the control is passed out of **try** by means of a jumping command. We will now, in Program 36.14 study an example of a try-catch-finally statement.

`Main` of Program 36.14 arranges that `M` is called (in line 30) for each value in the enumeration type `Control` (line 6). Inside the static method `M` we illustrate a number of possible ways out of `M`:

1. If `reason` is `Returning` , `M` calls **return** .
2. If `reason` is `Jumping` , `M` calls **goto** which brings the control outside try-catch-finally.
3. If `reason` is `Continue` , **continue** forces the for loop to the next iteration, which is non-existing. The call of **continue** leaves the try-catch-finally abruptly.
4. If `reason` is `Breaking` , **break** breaks out of the for loop, and try-catch-finally is left abruptly.
5. If `reason` is `Throwing` , an `Exception` is thrown. The exception is "handled" locally.
6. If `reason` is 5, the expression `(Control)i` does not hit a value in the `Control` enumeration type. This causes execution and termination of the try clause, in particular the execution of `WriteLine` in line 17.

```
1  using System;
2  
3  class FinallyDemo{
4  
5    internal enum Control {Returning, Jumping, Continuing, Breaking,
6                           Throwing, Normal}
7  
8    public static void M(Control reason){
9      for(int i = 1; i <= 1; i++)  // a single iteration
10       try{
11         Console.WriteLine("\nEnter try: {0}", reason);
12         if (reason == Control.Returning) return;
13         else if (reason == Control.Jumping) goto finish;
14         else if (reason == Control.Continuing) continue;
15         else if (reason == Control.Breaking) break;
16         else if (reason == Control.Throwing) throw new Exception();
17         Console.WriteLine("Inside try");
18       }
19       catch(Exception){
20         Console.WriteLine("Inside catch");
21       }
22       finally{
23         Console.WriteLine("Inside finally");
24       }
25     finish: return;
```

```
  }

  public static void Main(){
    for(int i = 0; i <= 5; i++)
      M((Control)i);
  }
}
```

Program 36.14 *Illustration of try-catch-finally.*

The outcome of the example in Program 36.14 can be seen in the program output in Listing 36.15. So please take a careful look at it.

```
Enter try: Returning
Inside finally

Enter try: Jumping
Inside finally

Enter try: Continuing
Inside finally

Enter try: Breaking
Inside finally

Enter try: Throwing
Inside catch
Inside finally

Enter try: Normal
Inside try
Inside finally
```

Listing 36.15 *Output from the try-catch-finally program.*

As it appears, the finally clause is executed in each of the six cases enumerated above. Thus, it is not possible to bypass a finally clause of a try-catch-finally control structure. The finally clause is executed independent of the way we execute and leave the try clause.

You should place code in **finally** clauses of try-catch-finally or try-finally which should be executed in all cases, both in case or "normal execution", in case of errors, and in case of exit-attempts via jumping commands.

36.10. Rethrowing an exception

Lecture 9 - slide 25

We will now study the idea of rethrowing an exception. We have already encountered and discussed exception rethrowing in Section 36.6 (see Program 36.4).

- Rethrowing
  - Preserving information about the original exception, and the call chain
  - Usually recommended

In Program 36.16 we illustrate rethrowing by means of the standard example of this chapter. The situation is as follows:

1. `Main` calls `M` , `M` calls `N` , `N` calls `P` .
2. In `P` an `IndexOutOfRangeException` exception is thrown as usual.
3. On its way back the calling chain, the exception is caught in `N` . But `N` regrets, and *rethrows* the exception.
4. The exception is passed unhandled through `M` and `Main` .

```
using System;

class ExceptionDemo{

  public static void Main(){
    Console.WriteLine("Main");
    int[] table = new int[6]{10,11,12,13,14,15};
    int idx = 6;
    M(table, idx);
  }

  public static void M(int[] table, int idx){
    Console.WriteLine("M(table,{0})", idx);
    N(table,idx);
  }

  public static void N(int[] table, int idx){
    Console.WriteLine("N(table,{0})", idx);
    try{
      P(table,idx);
    }
    catch (IndexOutOfRangeException e){
      // Will not/cannot handle exception here.
      // Rethrow original exception.
      throw;
    }
  }

  public static void P(int[] table, int idx){
    Console.WriteLine("P(table,{0})", idx);
    Console.WriteLine("Accessing element {0}: {1}",
                       idx, table[idx]);
  }
}
```

Program 36.16 *Rethrowing an exception.*

The output of Program 36.16 is shown in Listing 36.17. From the stack trace in Listing 36.17 it does not appear that the static method `N` actually has touched (and "smelled to") the exception. This is a main point of this example.

```
Main
M(table,6)
N(table,6)
P(table,6)

Unhandled Exception:
 System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at ExceptionDemo.P(Int32[] table, Int32 idx)
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.17 *Output from the program that rethrows an exception.*

Touching, but not handling the exception

An outer handler will see the original exception

36.11. Raising an exception in an exception handler

Lecture 9 - slide 26

We will now study an alternative to rethrowing, as discussed and illustrated in Program 36.16.

- Raising and throwing a new exception
    - Use this approach if you, of some reason, want to hide the original exception
        - Security, simplicity, ...
    - Consider propagation of the inner exception

In Program 36.18 we show a program similar to the program discussed in the previous section. Instead of rethrowing the exception in `N`, we throw a new instance of `IndexOutOfRangeException`. As can be seen in Listing 36.19 this affects the stack trace. From the outside, we can no longer see that the problem occurred in P.

```
using System;

class ExceptionDemo{

  public static void Main(){
    int[] table = new int[6]{10,11,12,13,14,15};
    int idx = 6;
    M(table, idx);
  }

  public static void M(int[] table, int idx){
    Console.WriteLine("M(table,{0})", idx);
    N(table,idx);
  }

  public static void N(int[] table, int idx){
    Console.WriteLine("N(table,{0})", idx);
    try{
      P(table,idx);
    }
    catch (IndexOutOfRangeException e){
      // Will not/can no handle here. Raise new exception.
      throw new IndexOutOfRangeException("Index out of range");
    }
  }

  public static void P(int[] table, int idx){
    Console.WriteLine("P(table,{0})", idx);
    Console.WriteLine("Accessing element {0}: {1}",
                       idx, table[idx]);
  }
}
```

Program 36.18 *Raising and throwing a new exception.*

```
M(table,6)
N(table,6)
P(table,6)

Unhandled Exception: System.IndexOutOfRangeException:
 Index out of range
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.19 *Output from the program that raises a new exception.*

As a final option, we may wish to reflect, in relation to the client, that the problem actually occurred in `P`. This can be done by passing an *inner exception* to the new exception, as constructed in line 24 of Program 36.20. Notice the effect this has on the stack trace in Listing 36.21.

```
1  using System;
2  
3  class ExceptionDemo{
4  
5    public static void Main(){
6      int[] table = new int[6]{10,11,12,13,14,15};
7      int idx = 6;
8      M(table, idx);
9    }
10 
11   public static void M(int[] table, int idx){
12     Console.WriteLine("M(table,{0})", idx);
13     N(table,idx);
14   }
15 
16   public static void N(int[] table, int idx){
17     Console.WriteLine("N(table,{0})", idx);
18     try{
19       P(table,idx);
20     }
21     catch (IndexOutOfRangeException e){
22       // Will not/cannot handle exception here.
23       // Raise new exception with propagation of inner exception.
24       throw new IndexOutOfRangeException("Index out of range",  e);
25     }
26   }
27 
28   public static void P(int[] table, int idx){
29     Console.WriteLine("P(table,{0})", idx);
```

```
    Console.WriteLine("Accessing element {0}: {1}",
                       idx, table[idx]);
  }
}
```

Program 36.20 *Raising and throwing a new exception, propagating original inner exception.*

```
M(table,6)
N(table,6)
P(table,6)

Unhandled Exception: System.IndexOutOfRangeException:
 Index out of range ---> System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
   at ExceptionDemo.P(Int32[] table, Int32 idx)
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   --- End of inner exception stack trace ---
   at ExceptionDemo.N(Int32[] table, Int32 idx)
   at ExceptionDemo.M(Int32[] table, Int32 idx)
   at ExceptionDemo.Main()
```

Listing 36.21 *Output from the program that raises a new exception, with inner exception.*

36.12. Recommendations about exception handling

Lecture 9 - slide 29

We are now almost done with exception handling. We will now formulate a few recommendations that are related to exception handling.

- Control flow
  - Do not use throw and try-catch as iterative or conditional control structures
  - Normal control flow should be done with normal control structures
- Efficiency
  - It is time consuming to throw an exception
  - It is more efficient to deal with the problem as a normal program aspect - if possible
- Naming
  - Suffix names of exception classes with "`Exception`"
- Exception class hierarchy
  - Your own exception classes should be subclasses of `ApplicationException`
  - Or alternatively (as of a more recent recommendation) of `Exception`.

- Exception classes
  - Prefer predefined exception classes instead of programming your own exception classes
  - Consider specialization of existing and specific exception classes
- Catching
  - Do not catch exceptions for which there is no cure
  - Leave such exceptions to earlier (outer) parts of the call-chain
- Burying
  - Avoid empty handler exceptions - exception burrying
  - If you touch an exception without handling it, always rethrow it

36.13. References

[Exceptions-best-practices]	Best practices for handling exceptions (MSDN) http://msdn.microsoft.com/en-us/library/seyhszts.aspx

37. Streams

We are now about to start the first chapter in the lecture about Input and Output (IO). Traditionally, IO deals with transfer of data to/from secondary storage, most notably disks. IO also covers the transmission of data to/from networks.

In this and the following chapters we will study the classes that are related to input and output. This includes file and directory classes. At the abstract level, the `Stream` class is the most important class in the IO landscape. Therefore we choose to start the IO story with an exploration of streams, and an understanding of the `Stream` class in C#. This includes several `Stream` subclasses and several client classes of `Stream`. The clients we have in mind are the so-called reader and writer classes.

37.1. The Stream Concept

Lecture 10 - slide 2

A stream is an abstract concept. A stream is a *connection* between a program and a storage/network. Essentially, we can read data from the stream into a program, or we can write data from a program to the stream. This understanding of a stream is illustrated in Figure 37.1.

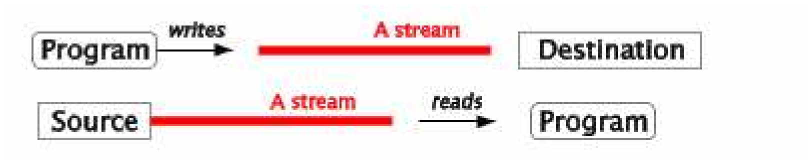

Figure 37.1 *Reading from and writing to a stream*

A *stream* is a flow of data from a program to a backing store, or from a backing store to a program

The program can either *write* to a stream, or *read* from a stream.

Stream and stream processing includes the following:

- Reading from or writing to files in secondary memory (disk)
- Reading from or writing to primary memory (RAM)
- Connection to the Internet
- Socket connection between two programs

The second item (reading and writing to/from primary memory) seems to be special compared to the others. Sometimes it may be attractive to have files in primary memory, and therefore it is natural that we should be able to use stream operation to access such files as well. In other situations, we wish to use internal data structures as sources or destinations of streams. It is, for instance, typical that we wish to read and write data from/to strings. We will see how this can be done in Section 37.14.

37.2. The abstract class Stream in C#

Lecture 10 - slide 3

The `Stream` class in C# is an abstract class (see Section 30.1). It belongs to the `System.IO` namespace, together with a lot other IO related types. The abstract `Stream` class provides a generic view on different kinds of sources and destinations, and it isolates client classes from the operating system details of these.

The `Stream` class supports both synchronous and asynchronous IO operations. Client classes that invoke a synchronous operation wait until the operation is completed before they can initiate other operations or actions. Use of a synchronous operation is not a problem if the operation is fast. Many IO operations on secondary storage are, however, very slow seen relative to the speed of the operations on primary storage. Therefore it may in some circumstances be attractive to initiate an IO operation, do something else, and consult the result of the IO operation at a later point in time. In order to provide for this, the `Stream` class supports the asynchronous IO operations `BeginRead` and `BeginWrite`. In the current version of the material we do not cover the asynchronous operations.

Let us now look at the most important operations on streams. The *`italic names`* refer to abstract methods. The abstract methods will be implemented in non-abstract subclasses of `Stream`.

- `int` *`Read`* `(byte[] buf, int pos, int len)`
- `int ReadByte()`
- `void` *`Write`* `(byte[] buf, int pos, int len)`
- `void WriteByte(byte b)`
- `bool` *`CanRead`*
- `bool` *`CanWrite`*
- `bool` *`CanSeek`*
- `long` *`Length`*
- `void` *`Seek`* `(long offset, SeekOrigin org)`
- `void` *`Flush`* `()`
- `void Close()`

In order to use `Read` you should allocate a byte array and pass (a reference to) this array as the first parameter of `Read`. The call `Read(buf, p, lgt)` reads *at most* `lgt` bytes, and stores them in `buf[p]` ... `buf[p+lgt-1]`. `Read` returns the actual number of characters read, which can be less than `lgt`.

`Write` works in a similar way. We assume that a number of bytes are stored in an existing byte array called `buf`. The call `Write(buf, p, lgt)` writes `lgt` bytes, `buf[p]` ... `buf[p+lgt-1]`, to the stream.

As you can see, only `ReadByte` and `WriteByte` are non-abstract methods. `ReadByte` returns the integer value of the byte being read, or -1 in case that the end of the stream has bee encountered. The two operations `ReadByte` and `WriteByte` rely on `Read` and `Write`. Internally, `ReadByte` calls `Read` on a one-byte array, it accesses this byte, and it returns this byte. `WriteByte` works in a similar way. Based on these informations, it is not surprising that it is recommended to redefine `ReadByte` and `WriteByte` in specialized `Stream` classes. The default implementations of `ReadByte` and `WriteByte` are simply too inefficient. The redefinitions should be able to profit from internal buffering.

The explanations of `Read` in relation to `ReadByte` (and `Write` in relation to `WriteByte`) may seem a little surprising. Why not have `ReadByte` as an abstract method, and `Read` as a non-abstract method, which once and for all is implemented in class `Stream` by multiple calls of `ReadByte`? Such a design seems to be ideal:

The task of implementing `ReadByte` in subclasses is easy, and no subclass should ever need to implement `Read`. The reason behind the actual design of the abstract `Stream` class is - of course - *efficiency*. The basic read and write primitives of streams should provide for efficient reading and writing. It is typically inefficient to read a single byte from a file. On many types of hardware (such as harddisks) we always read many bytes at a time. The design of the read and write operations of stream take advantage of this observation.

It is not possible to read, write, and seek in all streams. Therefore it is possible to query a stream for its actual capabilities. The boolean operations (properties) `CanRead`, `CanWrite`, `CanSeek` are used for such querying.

The static field **`Null`** represents a stream without a backing store.

`Null` is a public static field of type `Stream` in the abstract class `Stream`. If you, for some reason, wish to discard the data that you write, you can write it to `Stream.Null`. You can also read from `Stream.Null`; This will always give zero as result, however.

37.3. Subclasses of class Stream

Lecture 10 - slide 5

The abstract class `Stream` is the superclass of a number of non-abstract classes. Below we list the most important of these. Like the class `Stream`, many of the subclasses of `Stream` belong to the `System.IO` namespace.

- `System.IO.FileStream`
  - Provides a stream backed by a file from the operating system
- `System.IO.BufferedStream`
  - Encapsulates buffering around another stream
- `System.IO.MemoryStream`
  - Provides a stream backed by RAM memory
- `System.Net.Sockets.NetworkStream`
  - Encapsulates a socket connection as a stream
- `System.IO.Compression.GZipStream`
  - Provides stream access to compressed data
- `System.Security.Cryptography.CryptoStream`
  - `Write` encrypts and `Read` decrypts
- *And others...*

We show example uses of class `FileStream` in Section 37.4 and Section 37.6. Please notice, however, that file IO is typically handled through one of the reader and writer classes, which behind the scene delegates the work to a `Stream` class. We have a lot more to say about the reader and writer classes later in this material. Section 37.9 will supply you with an overview of the reader and writer classes in C#.

The class `BufferedStream` is intended to be used as a so-called decorator of another stream class. In Section 40.1 we discuss the ***Decorator*** design pattern. The concrete example of ***Decorator***, which we will discuss in Section 40.2, involves compressed streams. Notice that it is not relevant to use buffering on `FileStream`, because it natively makes use of buffering.

37.4. Example: Filestreams

Lecture 10 - slide 6

`FileStream` IO, as illustrated by the examples in this section, is used for *binary input and output*. It means that the `FileStream` operations transfer raw chuncks of bits between the program and the file. The bits are not interpreted. As a contrast, the reader and writer classes introduced in Section 37.9 interpret and transforms the raw binary data to values in C# types.

Let us show a couple of very simple programs that write to and read from filestreams. Figure 37.1 writes bytes corresponding to the three characters 'O', 'O', and 'P' to the file `myFile.bin`. Notice that we do not write characters, but numbers that belong to the simple type `byte`. The file opening is done via construction of the `FileStream` object in `Create` mode. `Create` is a value in the enumeration type `FileMode` in the namespace `System.IO`. File closing is done by the `Close` method.

```
using System.IO;

class ReadProg {
  static void Main() {
    Stream s = new FileStream("myFile.bin", FileMode.Create);
    s.WriteByte(79);  // O    01001111
    s.WriteByte(79);  // O    01001111
    s.WriteByte(80);  // P    01010000
    s.Close();
  }
}
```

Program 37.1 *A program that writes bytes corresponding to 'O' 'O' 'P' to a file stream.*

After having executed the program in Figure 37.1 the file `myFile.bin` exists. Program 37.2 reads it. We create a `FileStream` object in `Open` mode, and we read the individual bytes with use of the `ReadByte` method. In line 11 and 12 we illustrate what happens if we read beyond the end of the file. We see that `ReadByte` in that case returns -1. The number -1 is not a value in type `byte`, which supports the range 0..255. Therefore the type of the value returned by `ReadByte` is `int`.

```
1  using System;
2  using System.IO;
3
4  class WriteProg {
5    static void Main() {
6      Stream s = new FileStream("myFile.bin", FileMode.Open);
7      int i, j, k, m, n;
8      i = s.ReadByte();  // O   79  01001111
9      j = s.ReadByte();  // O   79  01001111
10     k = s.ReadByte();  // P   80  01010000
11     m = s.ReadByte();  // -1  EOF
12     n = s.ReadByte();  // -1  EOF
13
14     Console.WriteLine("{0} {1} {2} {3} {4}", i, j, k, m, n);
15     s.Close();
16   }
17 }
```

Program 37.2 *A program that reads the written file.*

37.5. The using control structure

Lecture 10 - slide 7

The simple file reading and writing examples in Section 37.4 show that file opening (in terms of creating the `FileStream` object) and file closing (in terms of sending a `Close` message to the stream) appear in pairs. This inspires a new control structure which ensures that the file always is closed when we are done with it. The syntax of the **using** construct is explained below.

```
using (type variable = initializer)
  body
```

Syntax 37.1 *The syntax of the using statement C#*

The meaning (semantics) of the using construct is the following:

- In the scope of **`using`**, bind *`variable`* to the value of *`initializer`*
- The *`type`* must implement the interface **`IDisposable`**
- Execute *body* with the established name binding
- At the end of *body* do *variable*`.Dispose`
    - The `Dispose` methods in the subclasses of `Stream` call `Close`

We encountered the interface `IDisposable` when we studied the interfaces in the C# libraries, see Section 31.4. The interface `IDisposable` prescribes a single method, `Dispose`, which in general is supposed to release resources. The abstract class `Stream` implements `IDisposable`, and the `Dispose` method of class `Stream` calls the Stream `Close` method.

Program 37.3 is a reimplementation of Program 37.1 that illustrates the **using** construct. Notice that we do not explicitly call `Close` in Program 37.3.

```
using System.IO;

class ReadProg {
  static void Main() {
    using(Stream s = new FileStream("myFile.txt", FileMode.Create)){
      s.WriteByte(79);  // O   01001111
      s.WriteByte(79);  // O   01001111
      s.WriteByte(80);  // P   01010000
    }
  }
}
```

Program 37.3 *The simple write-program programmed with 'using'.*

The following fragment shows what is actually covered by a **using** construct. Most important, a **try-finally** construct is involved, see Section 36.9. The use of **try-finally** implies that `Dispose` will be called independent of the way we leave `body`. Even if we attempt to exit `body` with a jump or via an exception, `Dispose` will be called.

```
// The using statement ...

  using (type variable = initializer)
    body

// ... is equivalent to the following try-finally statement

  {type variable = initializer;
    try {
      body
    }
    finally {
      if (variable != null)
         ((IDisposable)variable).Dispose();
    }
  }
```

Program 37.4 *The control structure 'using' defined by 'try-finally'.*

37.6. More FileStream Examples

Lecture 10 - slide 8

We will show yet another simple example of `FileStreams`, namely a static method that copies one file to another.

```
using System;
using System.IO;

public class CopyApp {

  public static void Main(string[] args) {
    FileCopy(args[0], args[1]);
  }

  public static void FileCopy(string fromFile, string toFile){
    try{
      using(FileStream fromStream =
                       new FileStream(fromFile, FileMode.Open)){
        using(FileStream toStream  =
                         new FileStream(toFile, FileMode.Create)){
          int c;

          do{
            c = fromStream.ReadByte();
            if(c != -1) toStream.WriteByte((byte)c);
          } while (c != -1);
        }
      }
    }
    catch(FileNotFoundException e){
      Console.WriteLine("File {0} not found: ", e.FileName);
      throw;
    }
    catch(Exception){
      Console.WriteLine("Other file copy exception");
      throw;
    }
 }

}
```

Program 37.5 *A FileCopy method in a source file copy-file.cs - uses two FileStreams.*

Exercise 10.1. *A variant of the file copy program*

The purpose of this exercise is to train the use of the `Read` method in class `Stream`, and subclasses of class `Stream`.

Write a variant of the file copy program. Your program should copy the entire file into a byte array. Instead of the method `ReadByte` you should use the `Read` method, which reads a number of bytes into a byte array. (Please take careful look at the documentation of `Read` in class `FileStream` before you proceed). After this, write out the byte array to standard output such that you can make sure that the file is correctly read.

Are you able to read the entire file with a single call to `Read`? Or do you prefer to read chunks of a certain (maximum) size?

37.7. The class Encoding

Lecture 10 - slide 10

Before we study the reader and writer classes we will clarify one important topic, namely *encodings*.

The problem is that a byte (as represented by a value of type `byte`) and a character (as represented as value of type `char`) are two different things. In the old days they were basically the same, or it was at least straightforward to convert one to the other. In old days there were at most 256 different characters available at a given point in time (corresponding to a straightforward encoding of a single character in a single byte). Today, the datatype `char` should be able to represent a wide variety of different characters that belong to different alphabets in different cultures. We still need to represent a character by means of a number of bytes, because a byte is a fundamental unit in most software and in most digital hardware.

As a naive approach, we could go for the following solution:

> We want to be able to represent a maximum of, say, 200000 different characters. For this purpose we need $\log_2(200000)$ bits, which is 18 bits. If we operate in units if 8 bits (= one byte) we see that we need at least 3 bytes per characters. Most likely, we will go for 4 bytes per character, because it fits much better with the word length of most computers. Thus, the byte size of a text will now be four times the size of an ASCII text. This is not acceptable because it would bloat the representation of text files on secondary disk storage.

As of 2007, the Unicode standard defines more than 100000 different characters. Unicode organizes characters in a number of *planes* of up to 2^{16} (= 65536) characters. The Basic Multilingual Plane - BMP - contains the most common characters.

Encodings are invented to solve the problem that we have outlined above. An encoding is a mapping between values of type character (a *code point* number between 0 and 200000 in our case) to a sequence of bytes. The naive approach outlined above represents a simple encoding, in which we need 4 bytes even for the original ASCII characters. It is attractive, however, if characters in the original, 7-bit ASCII alphabet can be encoded in a single byte. The price of that may very well be that some rarely used characters will need considerable more bytes for their encoding.

Let us remind ourselves that in C#, the type `char` is represented as 16 bit entities (Unicode characters) and that a `string` is a sequence of values of type `char`. We have already touched on this in Section 6.1. At the time Unicode was designed, it was hypothesized that 16 bits was enough to to represent all characters in the world. As mentione above, this turned out not to be true. Therefore the type `char` in C# is not big enough to hold all Unicode characters. The remedy is to use multiple `char` values for representation of a single Unicode character. We see that history repeats itself...

An encoding is a mapping between characters/strings and byte arrays

An object of class **`System.Text.Encoding`** represents knowledge about a particular character encoding

Let us now review the operations in class `Encoding:`, which is located in the namespace `System.Text` :

- **`byte[] GetBytes(string)`** *Instance method*
- **`byte[] GetBytes(char[])`** *Instance method*
  - Encodes a string/char array to a byte array relative to the current encoding
- **`char[] GetChars(byte[])`** *Instance method*
  - Decodes a byte array to a char array relative to the current encoding
- **`byte[] Convert(Encoding, Encoding, byte[])`** *Static method*
  - Converts a byte array from one encoding (first parameter) to another encoding (second parameter)

The method `GetBytes` implements the encoding in the direction of characters to byte sequences. In concrete terms, the method `GetBytes` transforms a `String` or an array of `char`s to a `byte` array.

The inverse method, `GetChars` converts an array of bytes to the corresponding array of characters. On a given string `str` and for a given encoding `e` `e.GetChars(e.GetBytes(str))` corresponds to `str`.

For given encodings `e1` and `e2`, and for some given byte array `ba` supposed to be encoded in `e1`, `Convert(e1,e2,ba)` is equivalent to `e2.GetBytes(e1.GetChars(ba))`.

37.8. Sample use of class Encoding

Lecture 10 - slide 11

Now that we understand the idea behind encodings, let us play a little with them. In Program 37.6 we make a number different encodings, and we convert a given string to some of these encodings. We explain the details after the program.

```
1  using System;
2  using System.Text;
3
4  /* Adapted from an example provided by Microsoft */
5  class ConvertExampleClass{
6    public static void Main(){
7      string unicodeStr =          // "A æ u å æ ø i æ å"
8          "A \u00E6 u \u00E5 \u00E6 \u00F8 i \u00E6 \u00E5";
9
10     // Different encodings.
11     Encoding ascii = Encoding.ASCII,
12              unicode = Encoding.Unicode,
13              utf8 = Encoding.UTF8,
14              isoLatin1 = Encoding.GetEncoding("iso-8859-1");
15
16     // Encodes the characters in a string to a byte array:
17     byte[] unicodeBytes = unicode.GetBytes(unicodeStr),
18            asciiBytes =   ascii.GetBytes(unicodeStr),
19            utf8Bytes =   utf8.GetBytes(unicodeStr),
20            isoLatin1Bytes =   utf8.GetBytes(unicodeStr);
21
22     // Convert from byte array in unicode to byte array in utf8:
23     byte[] utf8BytesFromUnicode =
24       Encoding.Convert(unicode, utf8, unicodeBytes);
25
26     // Convert from byte array in utf8 to byte array in ascii:
27     byte[] asciiBytesFromUtf8 =
28       Encoding.Convert(utf8, ascii, utf8Bytes);
29
30     // Decodes the bytes in byte arrays to a char array:
31     char[] utf8Chars = utf8.GetChars(utf8BytesFromUnicode);
32     char[] asciiChars = ascii.GetChars(asciiBytesFromUtf8);
33
34     // Convert char[] to string:
35     string utf8String = new string(utf8Chars),
36            asciiString = new String(asciiChars);
37
38     // Display the strings created before and after the conversion.
39     Console.WriteLine("Original string: {0}", unicodeStr);
40     Console.WriteLine("String via UTF-8: {0}", utf8String);
41
42     Console.WriteLine("Original string: {0}", unicodeStr);
43     Console.WriteLine("ASCII converted string: {0}", asciiString);
44   }
45 }
```

Program 37.6 *Sample encodings, conversions, and decodings of a string of Danish characters.*

In line 7 we declare a sample string, `unicodeStr`, which we initialize to a string with plenty of national Danish characters. We notate the string with escape notation `\u`*dddd* where *d* is a hexadecimal digit. We could, as well, have used the string constant in the comment at the end of line 7.

In line 11-14 we make a number of instances of class `Encoding`. Some common `Encoding` objects can be accessed conveniently via static properties of class `Encoding`. The UTF-8 encoding can in that way be accessed with `Encoding.UTF8`. The static method `GetEncoding` accesses an encoding via the name of the encoding. (In order to get access to all supported encodings, the static method `GetEncodings` (plural) is useful). The ISO Latin 1 encoding is accessed via use with use of `GetEncoding` in line 14.

In line 17-20 we convert the string `unicodeStr` to `byte` arrays in different encodings. For this purpose we use the instance method `GetBytes`.

Next, in line 22-28, we show how to use the static method `Convert` to convert a `byte` array in one encoding to a `byte` array in another encoding.

In line 30-32 it is shown how to convert byte arrays in a particular encoding to a char array. It is done by the instance method `GetChars`. We most probably wish to obtain a string instead of a `char` array. For that purpose we just use an appropriate `String` constructor, as shown in line 34-36.

In line 38-43 we display the values of `utf8String` and `asciiString`, and for comparison we also print the original `unicodeStr`. The printed result is shown in Listing 37.7. It is not surprising that the national Danish characters cannot be represented in the ASCII character set. The Danish characters are (ambiguously) translated to '?'.

```
Original string: A æ u å æ ø i æ å
String via UTF-8: A æ u å æ ø i æ å
Original string: A æ u å æ ø i æ å
ASCII converted string: A ? u ? ? ? i ? ?
```

Listing 37.7 *Output from the Encoding program.*

Exercise 10.2. *Finding the encoding of a given text file*

Make a UTF-8 text file with some words in Danish. Be sure to use plenty of special Danish characters. You may consider to write a simple C# program to create the file. You may also create the text file in another way.

In this exercise you should avoid writing a byte order mark (BOM) in your UTF-8 text file. (A BOM in the UTF-8 text file may short circuit the decoding we are asking for later in the exercise). One way to avoid the BOM is to denote the UTF-8 encoding with `new UTF8Encoding()`, or equivalently `new UTF8Encoding(false)`. You may want to consult the constructors in class `UFT8Encoding` for more information.

Now write a C# program which systematically - in a loop - reads the text file six times with the following objects of type `Encoding:` ISO-8859-1, UTF-7, UTF-8, UTF-16 (Unicode), UTF32, and 7 bits ASCII.

More concretely, I suggest you make a list of six encoding objects. For each encoding, open a `TextReader` and read the entire file (with `ReadToEnd`, for instance) with the current encoding. Echo the characters, which you read, to standard output.

You should be able to recognize the correct, matching encoding (UTF-8) when you see it.

37.9. Readers and Writers in C#

Lecture 10 - slide 9

In the rest of this chapter we will explore a family of so-called reader and writer classes. In most practical cases one or more of these classes are used for IO purposes instead of a `Stream` subclass, see Section 37.2.

Table 37.1 provides an overview of the reader and writer classes. In the horizontal dimension we have input (readers) and output (writers). In the vertical dimension we distinguish between binary (bits structured as bytes) and text (char/string) IO.

	Input	Output
Text	*TextReader* StreamReader StringReader	*TextWriter* StreamWriter StringWriter
Binary	BinaryReader	BinaryWriter

Table 37.1 *An overview of Reader and Writer classes*

The class `Stream` and its subclasses are oriented towards input and output of bytes. In contrast, the reader and writer classes are able to deal with input and output of characters (values of type char) and values of other simple types. Thus, the reader and writer classes operate at a higher level of abstraction than the stream classes.

In Section 37.3 we listed some important subclasses of class `Stream`. We will now discuss how the reader and writer classes in Table 37.1 are related to the stream classes. None of the classes in Table 37.1 inherit from class `Stream`. Rather, they *delegate* part of their work to a `Stream` class. Thus, the reader and writer classes aggregate (***have a***) `Stream` class together with other pieces of data. The class `StreamReader`, `StreamWriter`, `BinaryReader`, and `BinaryWriter` all have constructors that take a `Stream` class as parameter. In that way, it is possible to build such readers and writes on a `Stream` class.

`TextReader` and `TextWriter` in Table 37.1 are abstract classes. Their subclasses `StringReader` and `StringWriter` are build on strings rather than on streams. We have more to say about `StringReader` and `StringWriter` in Section 37.14.

In the following sections we will rather systematically describe the reader and writer classes in Table 37.1, and we will show examples of their use.

37.10. The class TextWriter

Lecture 10 - slide 12

In this section we discuss the abstract class `TextWriter`, and not least its non-abstract subclass `StreamWriter`. We cover the sibling classes `StringWriter` and `StringReader` in Section 37.14.

Most important, class `TextWriter` supports writing of text - characters and strings - via a chosen encoding. Encodings were discussed in Section 37.7. With use of class `TextWriter` it is also possible to write textual representations of simple types, such as `int` and `double`.

We illustrate the use of class `StreamWriter` in Program 37.8. Recall from Table 37.1 that `StreamWriter` is a non-abstract subclass of class `TextWriter`.

In Program 37.8 we write `str` and `strEquiv` (in line 9-10) to three different files. Both strings are identical, they contain a lot of Danish letters, but they are notated differently. It is the same string that we used in Program 37.6 for illustration of encodings. For each of the files we use a particular encoding (see Section 37.7). Notice that we in line 12, 16 and 20 use a `StreamWriter` constructor that takes a `Stream` and an encoding as parameters. There a six other constructors to chose from (see below). In line 24-26 we write the two strings to each of the three files. Try out the program, and read the three text files with your favorite text editor. Depending of the capabilities of your editor, you may or may not be able to read them all.

```
using System;
using System.IO;
using System.Text;

public class TextWriterProg{

  public static void Main(){
    string str =      "A æ u å æ ø i æ å",
           strEquiv = "A \u00E6 u \u00E5 \u00E6 \u00F8 i \u00E6 \u00E5";

    TextWriter
      tw1 = new StreamWriter(                        // Iso-Latin-1
             new FileStream("f-iso.txt", FileMode.Create),
             Encoding.GetEncoding("iso-8859-1")),

      tw2 = new StreamWriter(                        // UTF-8
             new FileStream("f-utf8.txt", FileMode.Create),
             new UTF8Encoding()),

      tw3 = new StreamWriter(                        // UTF-16
             new FileStream("f-utf16.txt", FileMode.Create),
             new UnicodeEncoding());

    tw1.WriteLine(str);     tw1.WriteLine(strEquiv);
    tw2.WriteLine(str);     tw2.WriteLine(strEquiv);
    tw3.WriteLine(str);     tw3.WriteLine(strEquiv);

    tw1.Close();
    tw2.Close();
    tw3.Close();
  }

}
```

Program 37.8   *Writing a text string using three different encodings with StreamWriters.*

You may wonder if knowledge about the applied encoding is somehow represented in the text file. The first few bytes in a text file created from a `TextWriter` may contain some information about the encoding. `StreamWriter` calls `Encoding.GetPreamble()` in order to get a byte array that represents knowledge about the encoding. This byte array is written in the beginning of the text file. This preamble is primarily used to determine the byte order of UTF-16 and UTF-32 encodings. (Two different byte orders are widely used on computers from different CPU manufacturers: Big-endian (most significant byte first) and little-endian (least significant byte first)). The preambles of the ASCII and the ISO Latin 1 encodings are empty.

The next program, shown in Program 37.9, first creates a `StreamWriter` on a given file path (a text string) "`simple-types.txt`". The default encoding is used. (The default encoding is system/culture dependent. It can be accessed with the static property `Encoding.Default`). By use of the heavily overloaded `Write` method it writes an integer, a double, a decimal, and a boolean to the file.

Next, from line 15-18, it writes a `Point` and a `Die` to a text file named "`non-simple-types.txt`". As expected, the `ToString` method is used on the `Point` and the `Die` objects. The contents of the two text files are shown in Listing 37.10 (only on web) and Listing 37.11 (only on web).

```
1 using System;
2 using System.IO;
3
4 public class TextSimpleTypes{
5
6   public static void Main(){

    using(TextWriter tw = new StreamWriter("simple-types.txt")){
      tw.Write(5);  tw.WriteLine();
      tw.Write(5.5);  tw.WriteLine();
      tw.Write(5555M); tw.WriteLine();
      tw.Write(5==6); tw.WriteLine();
    }

    using(TextWriter twnst = new StreamWriter("non-simple-types.txt")){
      twnst.Write(new Point(1,2)); twnst.WriteLine();
      twnst.Write(new Die(6)); twnst.WriteLine();
    }

  }
}
```

Program 37.9 *Writing values of simple types and objects of our own classes.*

The following items summarize the operations in class `StreamWriter`:

- 7 overloaded constructors
  - Parameters involved: File name, stream, encoding, buffer size
  - **`StreamWriter(String)`**
  - **`StreamWriter(Stream)`**
  - **`StreamWriter(Stream, Encoding)`**
  - *others*
- 17/18 overloaded **`Write`** / **`WriteLine`** operations
  - Chars, strings, simple types. Formatted output
- **`Encoding`**
  - A property that gets the encoding used for this `TextWriter`
- **`NewLine`**
  - A property that gets/sets the applied newline string of this `TextWriter`
- *others*

Exercise 10.3. *Die tossing - writing to text file*

Write a program that tosses a `Die` 1000 times, and writes the outcome of the tosses to a textfile. Use a `TextWriter` to accomplish the task.

Write another program that reads the text file. Report the number of ones, twos, threes, fours, fives, and sixes.

37.11. The class TextReader

Lecture 10 - slide 15

The class `TextReader` is an abstract class of which `StreamReader` is a non-abstract subclass. `StreamReader` is able to read characters from a byte stream relative to a given encoding. In most respects, the class `TextReader` is symmetric to class `TextWriter`. However, there are no `Read` counterparts to all the overloaded `Write` methods in `TextWriter`. We will come back to this observation below.

Program 37.12 is a program that reads the text that was produced by Program 37.8. In Program 37.12 we create three `TextReader` object. They are all based on file stream objects and encodings similar to the ones used in Program 37.8. From each `TextReader` we read the two strings that we wrote in Program 37.8. It is hardly surprising that we get six instances of the strange string "A æ u å æ ø i æ å". In line 19-21 they are all written to standard output via use of `Console.WriteLine`.

The last half part of Program 37.12 (from line 27) reads the three files as binary information (as raw bytes). The purpose of this reading is to exercise the *actual contents* of the three files. This is done by opening each of the files via `FileStream` objects, see Section 37.4. Recall that `FileStream` allows for binary reading (in terms of bytes) of a file. The function `StreamReport` (line 39-49) reads each byte of a given `FileStream`, and it prints these bytes on the console. The output in Listing 37.13 reveals - as expected - substantial differences between the actual, binary contents of the three files. Notice that the ISO Latin 1 file is the shortest, the UTF-8 file is in between, and the UTF-16 file is the longest.

```
1  using System;
2  using System.IO;
3  using System.Text;
4
5  public class TextReaderProg{
6
7    public static void Main(){
8
9      TextReader tr1 = new StreamReader(
10                        new FileStream("f-iso.txt", FileMode.Open),
11                        Encoding.GetEncoding("iso-8859-1")),
12                tr2 = new StreamReader(
13                        new FileStream("f-utf8.txt", FileMode.Open),
14                        new UTF8Encoding()),
15                tr3 = new StreamReader(             // UTF-16
16                        new FileStream("f-utf16.txt", FileMode.Open),
17                        new UnicodeEncoding());
18
19     Console.WriteLine(tr1.ReadLine());  Console.WriteLine(tr1.ReadLine());
20     Console.WriteLine(tr2.ReadLine());  Console.WriteLine(tr2.ReadLine());
21     Console.WriteLine(tr3.ReadLine());  Console.WriteLine(tr3.ReadLine());
22
23     tr1.Close();
24     tr2.Close();
25     tr3.Close();
26
27     // Raw reading of the files to control the contents at byte level
28     FileStream  fs1 = new FileStream("f-iso.txt", FileMode.Open),
29                 fs2 = new FileStream("f-utf8.txt", FileMode.Open),
30                 fs3 = new FileStream("f-utf16.txt", FileMode.Open);
31
32     StreamReport(fs1, "Iso Latin 1");
33     StreamReport(fs2, "UTF-8");
34     StreamReport(fs3, "UTF-16");
35
36     fs1.Close();
37     fs2.Close();
38     fs3.Close();
39   }
40
41   public static void StreamReport(FileStream fs, string encoding){
42     Console.WriteLine();
43     Console.WriteLine(encoding);
44     int ch, i = 0;
45     do{
46       ch = fs.ReadByte();
47       if (ch != -1) Console.Write("{0,4}", ch);
```

```
      i++;
      if (i%10 == 0) Console.WriteLine();
    } while (ch != -1);
    Console.WriteLine();
  }

}
```

Program 37.12 *Reading back the text strings encoded in three different ways, with StreamReader.*

```
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å

Iso Latin 1
  65  32 230  32 117  32 229  32 230  32
 248  32 105  32 230  32 229  13  10  65
  32 230  32 117  32 229  32 230  32 248
  32 105  32 230  32 229  13  10

UTF-8
  65  32 195 166  32 117  32 195 165  32
 195 166  32 195 184  32 105  32 195 166
  32 195 165  13  10  65  32 195 166  32
 117  32 195 165  32 195 166  32 195 184
  32 105  32 195 166  32 195 165  13  10


UTF-16
 255 254  65   0  32   0 230   0  32   0
 117   0  32   0 229   0  32   0 230   0
  32   0 248   0  32   0 105   0  32   0
 230   0  32   0 229   0  13   0  10   0
  65   0  32   0 230   0  32   0 117   0
  32   0 229   0  32   0 230   0  32   0
 248   0  32   0 105   0  32   0 230   0
  32   0 229   0  13   0  10   0
```

Listing 37.13 *Output from the program that reads back the strings encoded in three different ways.*

Below, in Program 37.14, we show a program that reads the values from the file "`simple-types.txt`", as written by Program 37.9. Notice that we read a line at a time using the `ReadLine` method of `StreamReader`. `ReadLine` returns a string, which we parse by the static `Parse` methods in the structs `Int32`, `Double`, `Decimal`, and `Boolean` respectively. There are no dedicated methods in class `StreamReader` for reading the textual representations of integers, doubles, decimals, booleans, etc. The output of Program 37.14 is shown in Listing 37.15 (only on web).

```
using System;
using System.IO;

public class TextSimpleTypes{

  public static void Main(){

    using(TextReader twst = new StreamReader("simple-types.txt")){
      int i = Int32.Parse(twst.ReadLine());
      double d = Double.Parse(twst.ReadLine());
      decimal m = Decimal.Parse(twst.ReadLine());
      bool b = Boolean.Parse(twst.ReadLine());

      Console.WriteLine("{0} \n{1} \n{2} \n{3}", i, d, m, b);
    }

  }
}
```

Program 37.14 *A program that reads line of text and parses them to values of simple types.*

As we did for class `TextWriter` in Section 37.10 we summarize the operations in class `TextReader` below:

- 10 StreamReader constructors
    - Similar to the StreamWriter constructors
    - **`StreamReader(String)`**
    - **`StreamReader(Stream)`**
    - **`StreamReader(Stream, bool)`**
    - **`StreamReader(Stream, Encoding)`**
    - *others*
- **`int Read()`** Reads a single character. Returns -1 if at end of file
- **`int Read(char[], int, int)`** Returns the number of characters read
- **`int Peek()`**
- **`String ReadLine()`**
- **`String ReadToEnd()`**
- **`CurrentEncoding`**
    - A property that gets the encoding of this StreamReader

The method `Read` reads a single character; It returns -1 if the file is positioned at the end of the file. The `Read` method that accepts three parameters is similar to the `Stream` method of the same name, see Section 37.2. As such, it reads a number of characters into an already allocated char array (which is passed as the first parameter of `Read`). `Peek` reads the next available character without advancing the file position. You can use the method to look a little ahead of the actual reading. As we have seen, `ReadLine` reads characters until an end of line character is encountered. Similarly, `ReadToEnd` reads the rest of stream - from the current position until the end of the file - and returns it as a string. `ReadToEnd` is often convenient if you wish to get access to a text file as a (potentially large) text string.

37.12. The class BinaryWriter

Lecture 10 - slide 18

In this section we will study a writer class which produces binary data. As such, a binary writer is similar to a `FileStream` used in write access mode, see Section 37.4. The justification of `BinaryWriter` is, however, that it supports a heavily overloaded `Write` method just like the class `TextWriter` did. The `Write` methods can be applied on most simple data types. The `Write` methods of `BinaryWriter` produce binary data, not characters.

Encodings, see Section 37.7, played important roles for `TextReader` and `TextWriter`. Encodings only play a minimal role in `BinaryWriter`; Encodings are only used when we write characters to the binary file.

Below, in Program 37.16 we show a program similar to Program 37.9. We write four values of different simple types to a file with use of a `BinaryWriter`. In comments of the program we show the expected number of bytes to be written. With use of a `FileInfo` object (see Section 38.1) we check our expectations in line 18-19. The output of the program is 29, as expected.

```
1  using System;
2  using System.IO;
3  
4  public class BinaryWriteSimpleTypes{
5  
6    public static void Main(){
7      string fn = "simple-types.bin";
8  
9      using(BinaryWriter bw = 
10             new BinaryWriter(
11               new FileStream(fn, FileMode.Create))){
12       bw.Write(5);      // 4  bytes
13       bw.Write(5.5);    // 8  bytes
14       bw.Write(5555M);  // 16 bytes
15       bw.Write(5==6);   // 1  bytes
16     }
17 
18     FileInfo fi = new FileInfo(fn);
19     Console.WriteLine("Length of {0}: {1}", fn, fi.Length);
20 
21   }
22 }
```

Program 37.16 *Use of a BinaryWriter to write some values of simple types.*

The following operations are supplied by `BinaryWriter`:

- Two public constructors
  - **`BinaryWriter(Stream)`**
  - **`BinaryWriter(Stream, Encoding)`**
- 18 overloaded **`Write`** operations
  - One for each simple type
  - **`Write(char)`**, **`Write(char[])`**, and **`Write(char[], int, int)`** - use Encoding
  - **`Write(string)`** - use Encoding
  - **`Write(byte[])`** and **`Write(byte[], int, int)`**
- **`Seek(int offset, SeekOrigin origin)`**
- *others*

The second constructor allows for registration of an encoding, which is used if we write characters as binary data. The `Write` methods, which accepts an array as first parameter together with two integers as second and third parameters, write a section of the involved arrays.

Exercise 10.4. *Die tossing - writing to a binary file*

This exercise is a variant of the die tossing and file writing exercise based on text files.

Modify the program to use a `BinaryWriter` and a `BinaryReader`.

Take notice of the different sizes of the text file from the previous exercise and the binary file from this exercise. Explain your observations.

37.13. The class BinaryReader

Lecture 10 - slide 20

The class `BinaryReader` is the natural counterpart to `BinaryWriter`. Both of them deal with input from and output to binary data (in contrast to text in some given encoding).

The following program reads the binary file produced by Program 37.16. It produces the expected output, see Program 37.16 (only on web).

```
using System;
using System.IO;

public class BinaryReadSimpleTypes{

  public static void Main(){
    string fn = "simple-types.bin";

    using(BinaryReader br =
            new BinaryReader(
              new FileStream(fn, FileMode.Open))){

      int i = br.ReadInt32();
      double d = br.ReadDouble();
      decimal dm = br.ReadDecimal();
      bool b = br.ReadBoolean();

      Console.WriteLine("Integer i: {0}", i);
      Console.WriteLine("Double d: {0}", d);
      Console.WriteLine("Decimal dm: {0}", dm);
      Console.WriteLine("Boolean b: {0}", b);
    }

  }
}
```

Program 37.17 *Use of a BinaryReader to write the values written by means of the BinaryWriter.*

The following gives an overview of the operations in the class `BinaryReader`:

- Two public constructors
  - **`BinaryReader(Stream)`**
  - **`BinaryReader(Stream, Encoding)`**
- 15 individually name **`Read`***type* operations
  - **`ReadBoolean`**, **`ReadChar`**, **`ReadByte`**, **`ReadDouble`**, **`ReadDecimal`**, **`ReadInt16`**, ...
- Three overloaded **`Read`** operations
  - **`Read()`** and **`Read (char[] buffer, int index, int count)`** read characters - using Encoding
  - **`Read (bytes[] buffer, int index, int count)`** reads bytes

The most noteworthy observation is that there exist a large number of specifically named operations (such as `ReadInt32` and `ReadDouble`) through which it is possible to read the binary representations of values in simple types.

37.14. The classes StringReader and StringWriter

Lecture 10 - slide 22

`StringReader` is a non-abstract subclass of `TextReader`. Similarly, `StringWriter` is a non-abstract subclass of `TextWriter`. Table 37.1 gives you an overview of these classes.

The idea of `StringReader` is to use traditional stream/file input operations for string access, and to use traditional stream/file output operations for string mutation. Thus, relative to Figure 37.1 the source and destinations of reading and writing will be strings.

A `StringReader` can be constructed on a string. A `StringWriter`, however, cannot be constructed on a string, because strings are non-mutable in C#, see Section 6.4. Therefore a `StringWriter` object is constructed on an instance of `StringBuilder`.

In Program 37.19 we illustrate, in concrete terms, how to make a `StringWriter` on the `StringBuilder` referred by the variable `sb` (see line 9). In line 11-17 we iterate five times through the for loop, with increasing integer values in the variable `i`. In total, the textual representations of 20 simple values are written to the `StringBuilder` object. The content of the `StringBuilder` object is printed in line 19. The output of Program 37.19 is shown in Program 37.20 (only on web).

```
1  using System;
2  using System.IO;
3  using System.Text;
4
5  public class TextSimpleTypes{
6
7    public static void Main(){
8
9      StringBuilder sb = new StringBuilder();   // A mutable string
10
11     using(TextWriter tw = new StringWriter(sb)){
12       for (int i = 0; i < 5; i++){
13         tw.Write(5 * i);  tw.WriteLine();
14         tw.Write(5.5 * i);  tw.WriteLine();
15         tw.Write(5555M * i); tw.WriteLine();
16         tw.Write(5 * i == 6); tw.WriteLine();}
17     }
18 
19     Console.WriteLine(sb);
20 
21   }
22 }
```

Program 37.19   *A StringWriter program similar to the StreamReader program shown earlier.*

Symmetrically, we illustrate how to read from a string. In Program 37.21 we make a string `str` with broken lines in line 8-11. With use of a `StringReader` built on `str` we read an integer, a double, a decimal, and a boolean value. The output is shown in Program 37.22 (only on web).

```
1  using System;
2  using System.IO;
3  
4  public class TextSimpleTypes{
5  
6    public static void Main(){
7  
8      string str = "5" + "\n" +
9                   "5,5" + "\n" +
10                  "5555,0" + "\n" +
11                  "false";
12 
13     using(TextReader tr = new StringReader(str)){
14       int i = Int32.Parse(tr.ReadLine());
15       double d = Double.Parse(tr.ReadLine());
16       decimal m = Decimal.Parse(tr.ReadLine());
17       bool b = Boolean.Parse(tr.ReadLine());
18 
19       Console.WriteLine("{0} \n{1} \n{2} \n{3}", i, d, m, b);
20     }
21 
22   }
23 }
```

Program 37.21   *A StringReader program.*

The use of `StringWriter` and `StringReader` objects for accessing the characters in strings is an attractive alternative to use of the native `String` and `StringBuilder` operations. It is, in particular, attractive and convenient that we can switch from a file source/destination to a string source/destination. In that way existing file manipulation programs may be used directly as string manipulation programs. The only necessary modification of the program is a replacement of a `StreamReader` with `StringReader`, or a replacement of `StreamWriter` with a `StringWriter`.

Be sure to use the abstract classes `TextReader` and `TextWriter` as much as possible. You should only use `StreamReader`/`StringReader` and `StreamWriter`/`StringWriter` for instantiation purposes in the context of a constructor (such as line 11 of Program 37.19 and line 13 of Program 37.21).

37.15. The Console class

Lecture 10 - slide 23

We have used static methods in the `Console` class in almost all our programs. It is now time to examine the `Console` class a little closer. In contrast to most other IO related classes, the `Console` class resides in the `System` namespace, and not in `System.IO`. The `Console` class encapsulates three streams: *standard input*, *standard output*, and *standard error*. The static property `In`, of type `TextReader`, represents standard input. The static properties `Out` and `Error` represent standard output and standard error respectively, and they are both of type `TextWriter`. Recall in this context that `TextReader` and `TextWriter` are both abstract classes, see Section 37.9.

```
1  using System;
2  using System.IO;
3  
4  class App{
5  
6    public static void Main(string[] args){
7  
8       TextWriter standardOutput = Console.Out;
9       StreamWriter myOut = null,
10                   myError = null;
11  
12      if (args.Length == 2) {
13         Console.Out.WriteLine("Redirecting std output and error to files");
14         myOut = new StreamWriter(args[0]);
15         Console.SetOut(myOut);
16         myError = new StreamWriter(args[1]);
17         Console.SetError(myError);
18      } else {
19         Console.Out.WriteLine("Keeping standard output and error unchanged");
20      }
21  
22      // Output from this section of the program may be redirected
23      Console.Out.WriteLine("Text to std output - by Console.Out.WriteLine");
24      Console.WriteLine("Text to standard output -  by Console.WriteLine(...)");
25      Console.Error.WriteLine("Error msg - by Console.Error.WriteLine(...)");
26  
27      if (args.Length == 2) {
28        myOut.Close(); myError.Close();
29      }
30  
31      Console.SetOut(standardOutput);
32      Console.Out.WriteLine("Now we are back again");
33      Console.Out.WriteLine("Good Bye");
34    }
35 }
```

Program 37.23 *A program that redirects standard output and standard error to a file.*

In the program shown above it is demonstrated how to control standard output and standard error. If we pass two program arguments (`args` in line 6) to Program 37.23 we redirect standard output and standard error to specific files (instances of `StreamWriter`) in line 13-17. That is the main point, which we wish to illustrate in Program 37.23.

Below we supply an overview of the methods and properties of the `Console` class. The `Console` class is static. As such, all methods and properties in class `Console` are static. There will never be objects of type `Console` around. The `Console` class offers the following operations:

- Access to and control of **`in`**, **`out`**, and **`error`**
- **`Write`**, **`WriteLine`**, **`Read`**, and **`ReadLine`** methods
    - Shortcuts to **`out.Write`**, **`out.WriteLine`**, **`in.Read`**, and **`in.ReadLine`**
- Many properties and methods that control the underlying buffer and window
    - Size, colors, and positions
- Immediate, non-blocking input from the Console
    - The property **`KeyAvailable`** returns if a key is pressed (non-blocking)
    - **`ReadKey()`** returns info about the pressed key (blocking)
- Other operations
    - **`Clear()`**, **`Beep()`**, and **`Beep(int, int)`** methods.

38. Directories and Files

The previous chapter was about streams, and as such also about files. In this chapter we will deal with the properties of files beyond reading and writing. File copying, renaming, creation time, existence, and deletion represent a few of these. In addition to files we will also in this chapter discuss directories.

38.1. The File and FileInfo classes

Lecture 10 - slide 26

Two overlapping file-related classes are available to the C# programmer: `FileInfo` and `File`. Both classes belong to the namespace `System.IO`. Objects of class `FileInfo` represents a single file, created on the basis of the name or path of the file (which is a string). The class `File` contains static methods for file manipulation. Class `File` is static, see Section 11.12, and as such there can be no instances of class `File`. If you intend to write object-oriented programs with file manipulation needs it is recommended that you represent files as instances of class `FileInfo`.

Let us right away write a program which illustrates how to use instances of class `FileInfo` for representation of files. All aspects related to class `FileInfo` is shown in **purple** in Program 38.1.

```
using System;
using System.IO;

public class FileInfoDemo{

  public static void Main(){
    // Setting up file names
    string fileName = "file-info.cs",
           fileNameCopy = "file-info-copy.cs";

    // Testing file existence
    FileInfo fi = new FileInfo(fileName); // this source file
    Console.WriteLine("{0} does {1} exist",
                      fileName, fi.Exists ? "" : "not");

    // Show file info properties:
    Console.WriteLine("DirectoryName: {0}", fi.DirectoryName);
    Console.WriteLine("FullName: {0}", fi.FullName);
    Console.WriteLine("Extension: {0}", fi.Extension);
    Console.WriteLine("Name: {0}", fi.Name);
    Console.WriteLine("Length: {0}", fi.Length);
    Console.WriteLine("CreationTime: {0}", fi.CreationTime);

    // Copy one file to another
    fi.CopyTo(fileNameCopy);
    FileInfo fiCopy  = new FileInfo(fileNameCopy);

    // Does the copy exist?
    Console.WriteLine("{0} does {1} exist",
                      fileNameCopy, fiCopy.Exists ? "" : "not");

    // Delete the copy again
    fiCopy.Delete();

    // Does the copy exist?
    Console.WriteLine("{0} does {1} exist",
                      fileNameCopy, fiCopy.Exists ? "" : "not"); // !!??
38
39      // Create new FileInfo object for the copy
40      FileInfo fiCopy1  = new FileInfo(fileNameCopy);
41      // Check if the copy exists?
42      Console.WriteLine("{0} does {1} exist", fileNameCopy,
43                        fiCopy1.Exists ? "" : "not");
44
45      // Achieve a TextReader (StreamReader) from the file info object
46      // and echo the lines in the file to standard output
47      using(StreamReader sr = fi.OpenText()){
48        for (int i = 1; i <= 10; i++)
49          Console.WriteLine("  " + sr.ReadLine());
50      }
51    }
52  }
```

Program 38.1   *A demonstration of the FileInfo class.*

In line 12 we create a `FileInfo` object on the source file of the C# program text shown in Program 38.1. In line 13-14 we report on the existence of this file in the file system. (We expect existence, of course). In line 16-22 we access various properties (in the sense of C# properties, see Chapter 18) of the `FileInfo` object. In line 25 we copy the file, and in line 30 we check the existence of the copy. In line 33 we delete the copy, and in line 37 we check the existence of copy again. Against our intuition, we find out that the copy of the file still exists after its deletion. (See next paragraph for an explanation). If, however, we establish a fresh `FileInfo` object on the path to the deleted file, we get the expected result. In line 45-50 we use the `OpenText` method of the `FileInfo` object to establish a `TextReader` on the file. Via a number of `ReadLine` activations in line 49 we demonstrate that we can read the contents of the file.

The file existence problem described above occurs because the instance of class `FileInfo` and the state of the underlying file system become inconsistent. The instance method `Refresh` of class `FileInfo` can be used to update the `FileInfo` object from the information in the operating system. If you need trustworthy information about your files, you should always call the `Refresh` operation before you access any `FileInfo` attribute. If we call `fiCopy.Refresh()` in line 34, the problem observed in line 37 vanishes.

The output of Program 38.1 is shown in Listing 38.2 (only on web).

The following gives an overview of some selected operations in class `FileInfo`:

- A single constructor
  - **`FileInfo(string)`**
- Properties (getters) that access information about the current file
  - Examples: **`Length`**, **`Extension`**, **`Directory`**, **`Exists`**, **`LastAccessTime`**
- Stream, reader, and writer *factory methods*:
  - Examples: **`Create`**, **`AppendText`**, **`CreateText`**, **`Open`**, **`OpenRead`**, **`OpenWrite`**, **`OpenText`**
- Classical file manipulations
  - **`CopyTo`**, **`Delete`**, **`MoveTo`**, **`Replace`**
- *Others*
  - **`Refresh`**, ...

The parameter of the `FileInfo` constructor is an absolute or relative path to a file. The file path must be *well-formed* according to a set of rules described in the class documentation. As examples, the file paths "`c:\temp c:\user`" and " `dir1\dir2\file.dat`" are both malformed.

We have also written af version of Program 38.1 in which we use the static class `File` instead of `FileInfo`, see Program 38.3. We do not include this program, nor the listing of its output, in the paper edition of the material. We notice that the file existence frustrations in Program 38.1 (of the deleted file) do not appear when we use the static operations of the static class `File`.

There is a substantial overlap between the instance methods of class **`FileInfo`** and the static methods in class **`File`**

38.2. The Directory and DirectoryInfo classes

Lecture 10 - slide 28

The classes `DirectoryInfo` and `Directory` are natural directory counterparts of the classes `FileInfo` and `File`, as described in Section 38.1. In this section we will show an example use of class `DirectoryInfo`, and we will provide an overview of the members in the class. Like for files, an instance of class `DirectoryInfo` is intended to represent a given directory of the underlying file system. We recommend that you use the class `DirectoryInfo`, rather than the static class `Directory`, when you write object-oriented programs.

It is worth noticing that the classes `FileInfo` and `DirectoryInfo` have a common abstract, superclass class `FileSystemInfo`.

Here follows a short program that use an instance of class `DirectoryInfo` for representation of a given directory from the underlying operating system.

```
using System;
using System.IO;

public class DirectoryInfoDemo{

  public static void Main(){
    string fileName = "directory-info.cs";    // The current source file

    // Get the DirectoryInfo of the current directory
    // from the FileInfo of the current source file
    FileInfo fi = new FileInfo(fileName);     // This source file
    DirectoryInfo di = fi.Directory;

    Console.WriteLine("File {0} is in directory \n   {1}", fi, di);

    // Get the files and directories in the parent directory.
    FileInfo[] files = di.Parent.GetFiles();
    DirectoryInfo[] dirs = di.Parent.GetDirectories();

    // Show the name of files and directories on the console
    Console.WriteLine("\nListing directory {0}:", di.Parent.Name);
    foreach(DirectoryInfo d in dirs)
      Console.WriteLine(d.Name);
    foreach(FileInfo f in files)
      Console.WriteLine(f.Name);

  }
}
```

Program 38.5 *A demonstration of the DirectoryInfo class.*

Like in Program 38.3 the starting point in Program 38.5 is a `FileInfo` object that represents the source file shown in Program 38.5. Based on the `FileInfo` object, we create a `DirectoryInfo` object in line 12. This `DirectoryInfo` object represents the directory in which the actual source file resides. Let us call it the *current directory* . In line 17 we illustrate the `Parent` property and the `GetFiles` method; We create an array, `files`, of `FileInfo` object of the parent directory of the current directory. Thus, this array holds all files of the parent of current directory. Similarly, `dirs` declared in line 18 is assigned to hold all directories of the parent of current directory. We print these files and directories in line 20-25.

The output of Program 38.5 (only on web) is shown in Listing 38.6 (only on web). A similar program, programmed with use of the static operations in class `Directory`, is shown in Program 38.7 (only on web).

The following shows an overview of the instance properties and instance methods in class **`DirectoryInfo`**:

- A single constructor
  - **`DirectoryInfo(string)`**
- Properties (getters) that access information about the current directory
  - Examples: **`CreationTime`**, **`LastAccessTime`**, **`Exists`**, **`Name`**, **`FullName`**
- Directory Navigation operations
  - Up: **`Parent`**, **`Root`**
  - Down: **`GetDirectories`**, **`GetFiles`**, **`GetFileSystemInfo`** (all overloaded)
- Classical directory manipulations
  - **`Create`**, **`MoveTo`**, **`Delete`**
- *Others*
  - **`Refresh`**, ...

The constructor takes a directory path string as parameter. It is possible to create a `DirectoryInfo` object on a string that represents a non-existing directory path. Like file paths, the given directory path must be well-formed (according to rules stated in the class documentation).

The downwards directory navigation operations `GetDirectories`, `GetFiles`, and `GetFileSystemInfo` are able to filter their results (with use of strings with wildcards, such as "`temp*`", which match all files/directories whose names start with "`temp`"). It is also possible to specify if the operations should access direct files/directories, or if they should access direct as well as indirect file/directories.

As for **`File`** and **`FileInfo`**, there is substantial overlap between the classes **`Directory`** and **`DirectoryInfo`**

39. Serialization

In this material we care about object-oriented programming. All our data are encapsulated in objects. When we deal with IO it is therefore natural to look for solutions that help us with *output and input of objects*.

For each class `C` it is possible to decide a *storage format*. The storage format of class `C` tells which pieces of data in `C` instances to save on secondary storage. The details of the storage format need to be decided. This involves (1) which fields to store, (2) the sequence of fields in the stored representation, and (3) use of a binary or a textual representation. However, as long as we have pairs of `WriteObject` and `ReadObject` operations for which `ReadObject(WriteObject(C-object))` is equivalent to `C-object` the details of the storage format are of secondary interest.

Instances of class `C` may have references to instances of other classes, say `D` and `E`. In general, an instance of class `C` may be part of an *object graph* in which we find `C`-object, `D`-object, `E`-objects as well as objects of other types. We soon realize that the real problem is not how to store instances of `C` in isolation. Rather, the problem is how to store an object network in which `C`-objects take part (or in which a `C`-object is a root).

People who have devised a storage format for a class `C`, who have written then `WriteObject` and `ReadObject` operations for class `C`, and who have dealt with the IO problem of object graphs quickly realize that the invented solutions generalizes to arbitrary classes. Thus, instead of solving the object IO problem again and again for specific classes, it is attractive to solve the problem at a general level, and make the solution available for arbitrary classes. This is exactly what serialization is about. The serialization problem has been solved by the implementers of C#. It is therefore easy for the C# programmer to save and retrieve objects via serialization.

39.1. Serialization

Lecture 10 - slide 31

Serialization provides for input and output of a network of objects. Serialization is about object output, and deserialization is about object input.

- Serialization
  - Writes an object *o* to a file
  - Also writes the objects referred from *o*
- Deserialization
  - Reads a serialized file in order to reestablish the serialized object *o*
  - Also reestablishes the network of objects originally referred from *o*

Serialization of objects is, in principle, simple to deal with from C#. There are, however, a couple of circumstances that complicate the matters:

- The need to control or customize the serialization and the deserialization of objects of specific types.
- The support of more than one C# technique to obtain the same serialization or deserialization effect.

The need to control (customize) the details of serialization and deserialization is unavoidable, at least when the ideas should be applied on real-life examples.

The support of several different techniques for doing serialization is due to the development of C#. In C# 2.0 serialization relies almost exclusively on the use of serialization and deserialization attributes. In C# 1.0 it was also necessary to implement certain interfaces to control and customize the serialization. In this version of the material, we only describe serialization controlled by attributes.

- Serialization and deserialization is supported via classes that implement the **`Iformatter`** interface:
    - `BinaryFormatter` and `SoapFormatter`
- Methods in `Iformatter`:
    - `Serialize` and `Deserialize`

In the following section we will discuss an example that uses `BinaryFormatter`.

39.2. Examples of Serialization in C#

Lecture 10 - slide 32

Below we show the class `Person` and class `Date`, similar to the ones we used for illustration of privacy leaks in Section 16.5. Class `Person` in Program 39.1 encapsulates a name and two date objects: birth date and death date. For a person still alive, the death date refer to `null`. *Redundantly*, the `age` instance variable holds the age of the person. The `Update` method can be used to update the `age` variable.

The `Date` class shown in Program 39.2 is a very simple implementation of a date class. (In the paper version of the material we only show an outline of the `Date` class. The complete version is available in the web version). The `Person` class relies on the `Date`. We use class `Date` for illustration of serialization; In real life you should always use the struct `DateTime`. The `Date` class encapsulates year, month, and day. In addition it holds a `nameOfDay` instance variable (with values such as `Sunday` or `Monday`), which is *redundant*. With appropriate calendar knowledge, the `nameOfDay` can be calculated from `year`, `month`, and `day`. The `Person` class needs age calculation, which is provided by the `YearDiff` method of class `Date`. Internally in class `Date`, `YearDiff` relies on the methods `IsBefore` and `Equals`. (`Equals` is defined according the standard recommendations, see Section 28.16. We have not, in this class, included a redefinition of `GetHashCode` and therefore we get a warning from the compiler when class `Date` is compiled.)

The *redundancy* is class `Person` and class `Date` is introduced on purpose, because it helps us illustrate the serialization control in Program 39.2. In most circumstances we would avoid such redundancy, at least in simple classes.

The preparation of class `Person` and class `Date` for serialization is very simple. We mark both classes with the attribute **`[Serializable]`**, see line 3 in both classes. As of now you can consider **`[Serializable]`** as some magic, special purpose notation. In reality **`[Serializable]`** represents application of an attribute. When we are done with serialization we have seen several uses of attributes, and therefore we will be motivated to understand the general ideas of attributes in C#. We discuss the general ideas behind attributes in Section 39.6.

Please notice that in the paper version of this material most program examples have been abbreviated. The full details of all examples appear in the web version of the material.

```
using System;

[Serializable]
public class Person{

  private string name;
  private int age;    // Redundant
  private Date dateOfBirth, dateOfDeath;

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

  public Date DateOfBirth {
    get {return new Date(dateOfBirth);}
  }

  public int Age{
    get {return Alive ? age : dateOfDeath.YearDiff(dateOfBirth);}
  }

  public bool Alive{
    get {return dateOfDeath == null;}
  }

  public void Died(Date d){
    dateOfDeath = d;
  }

  public void Update(){
    age = Date.Today.YearDiff(dateOfBirth);
  }

  public override string ToString(){
    return "Person: " + name +
            "  *" + dateOfBirth +
            (Alive ? "" : "  +" + dateOfDeath) +
            "  Age: " + age;
  }

}
```

Program 39.1 *The Person class - Serializable.*

```
using System;

[Serializable]
public class Date{
  private ushort year;
  private byte month, day;
  private DayOfWeek nameOfDay;    // Redundant

  public Date(int year, int month, int day){
    this.year =  (ushort)year;
    this.month = (byte)month;
    this.day =   (byte)day;
    this.nameOfDay = (new DateTime(year, month, day)).DayOfWeek;
  }

  public Date(Date d){
    this.year = d.year; this.month = d.month;
    this.day = d.day; this.nameOfDay = d.nameOfDay;
  }

  public int Year{get{return year;}}
  public int Month{get{return month;}}
  public int Day{get{return day;}}

  // return this minus other, as of usual birthday calculations.
  public int YearDiff(Date other){
    // ...
  }

  public override bool Equals(Object obj){
    // ...
  }

  // Is this date less than other date
  public bool IsBefore(Date other){
    // ...
  }

  public static Date Today{
    // ...
  }

  public override string ToString(){
    return string.Format("{0} {1}.{2}.{3}", nameOfDay, day, month, year);
  }
}
```

Program 39.2 *An outline of the Date class - Serializable.*

In Program 39.3 it is illustrated how to serialize and deserialize a graph of objects. The graph, which we serialize, consists of one `Person` and the two `Date` objects referred by the `Person` object. The serialization, which takes place in line 13-17, is done by sending the `Serialize` message to the `BinaryFormatter` object. The serialization relies on a binary stream, as represented by an instance of class `FileStream`, see Section 37.4.

The deserialization, as done in line 24-28, will in most real-life settings be done in another program. In our example we reset the program state in line 19-22 before the deserialization. The actual deserialization is done by sending the `Deserialize` message to the `BinaryFormatter` object. As in the serialization, the file stream with the binary data, is passed as a parameter.

```
1  using System;
2  using System.IO;
3  using System.Runtime.Serialization;
4  using System.Runtime.Serialization.Formatters.Binary;
5
6  class Client{
7
8    public static void Main(){
9      Person p = new Person("Peter", new Date(1936, 5, 11));
10     p.Died(new Date(2007,5,10));
11     Console.WriteLine("{0}", p);
12
13     using (FileStream strm =
14                new FileStream("person.dat", FileMode.Create)){
15       IFormatter fmt = new BinaryFormatter();
16       fmt.Serialize(strm, p);
17     }
18
19     // -----------------------------------------------------------
20     p = null;
21     Console.WriteLine("Reseting person");
22     // -----------------------------------------------------------
23
24     using (FileStream strm =
25                 new FileStream("person.dat", FileMode.Open)){
26       IFormatter fmt = new BinaryFormatter();
27       p = fmt.Deserialize(strm) as Person;
28     }
29
30     Console.WriteLine("{0}", p);
31   }
32
33 }
```

Program 39.3 *The Person client class - applies serialization and deserialization.*

The program output shown in Listing 39.4 tells that the `Person` object and the two `Date` objects have survived the serialization and deserialization processes. In between the two output lines in line 11 and line 30 of Program 39.3 the three objects have been transferred to and reestablished from the binary file.

```
Person: Peter  *Monday 11.5.1936  +Thursday 10.5.2007  Age: 71
Reseting person
Person: Peter  *Monday 11.5.1936  +Thursday 10.5.2007  Age: 71
```

Listing 39.4 *Output of the Person client class.*

Exercise 10.5. *Serializing with an XML formatter*

In the programs shown on the accompanying slide we have used a binary formatter for serialization of `Person` and `Date` object.

Modify the client program to use a so-called Soap formatter in the namespace `System.Runtime.Serialization.Formatters.Soap`. SOAP is an XML language intended for exchange of XML documents. SOAP is related to the discipline of web services in the area of Internet technology.

After the serialization you should take a look at the file `person.dat`, which is written and read by the client program.

39.3. Custom Serialization

Lecture 10 - slide 33

In the `Person` and `Date` classes, shown in Section 39.2, the redundant instance variables do not need to be serialized. In class `Person`, `age` does need to be serialized because it can be calculated from `dateOfBirth` and `dateOfDeath`. In class `Date`, `nameOfDay` does need to serialized because it can calculated from calendar knowledge. In relation to serialization and persistence, we say that these two instance variables are *transient*. It is sufficient to serialize the essential information, and to reestablish the values of the transient instance variables after deserialization. In Program 39.5 and Program 39.6 we show the serialization and the deserialization respectively.

The serialization is controlled by marking some fields (instance variables) as **[NonSerialized]**, see line 9 of Program 39.5 and line 9 of Program 39.6.

The deserialization is controlled by a method marked with the attribute **[OnDeserialized()]**, see line 21 of Program 39.5. This method is called when deserialization takes place. The method starting at line 21 of Program 39.5 assigns the redundant `age` variable of a `Person` object.

```
1  using System;
2  using System.Runtime.Serialization;
3  
4  [Serializable]
5  public class Person{
6  
7    private string name;
8  
9    [NonSerialized()]
10   private int age;
11 
12   private Date dateOfBirth, dateOfDeath;
13 
14   public Person (string name, Date dateOfBirth){
15     this.name = name;
16     this.dateOfBirth = dateOfBirth;
17     this.dateOfDeath = null;
18     age = Date.Today.YearDiff(dateOfBirth);
19   }
20 
21   [OnDeserialized()]
22   internal void FixPersonAfterDeserializing(
23                          StreamingContext context){
24     age = Date.Today.YearDiff(dateOfBirth);
25   }
26 
27   // ...
28 
29 }
```

Program 39.5 *The Person class - Serialization control with attributes.*

The `Date` class shown below in Program 39.6 follows the same pattern as the `Person` class of Program 39.5.

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class Date{
  private ushort year;
  private byte month, day;

  [NonSerialized()]
  private DayOfWeek nameOfDay;

  public Date(int year, int month, int day){
    this.year =  (ushort)year;
    this.month = (byte)month;
    this.day =   (byte)day;
    this.nameOfDay = (new DateTime(year, month, day)).DayOfWeek;
  }

  public Date(Date d){
    this.year = d.year; this.month = d.month;
    this.day = d.day; this.nameOfDay = d.nameOfDay;
  }

  [OnDeserialized()]
  internal void FixDateAfterDeserializing(
                                 StreamingContext context){
    nameOfDay = (new DateTime(year, month, day)).DayOfWeek;
  }

  // ...
}
```

Program 39.6 *The Date class - Serialization control with attributes .*

39.4. Considerations about Serialization

Lecture 10 - slide 34

We want to raise a few additional issues about serialization:

- Security
  - Encapsulated and private data is made available in files
- Versioning
  - The private state of class C is changed
  - It may not be possible to read serialized objects of type C
- Performance
  - Some claim that serialization is relatively slow

39.5. Serialization and Alternatives

Lecture 10 - slide 35

As mentioned in the introduction of this chapter - Chapter 39 - serialization deals with input and output of objects and object graphs. It should be remembered, however, that there are alternatives to serialization. As summarized below, it is possible to program object IO at a low level (using binary of textual IO primitives from Chapter 37). At the other end of the spectrum it is possible us database technology.

- Serialization
  - An easy way to save and restore objects in between program sessions
  - Useful in many projects where persistency is necessary, but not a key topic
  - Requires only little programming
- Custom programmed file IO
  - Full control of object IO
  - May require a lot of programming
- Objects in Relational Databases
  - *Impedance mismatch*: "Circular objects in retangular boxes"
  - Useful when the program handles large amounts of data
  - Useful if the data is accessed simultaneous from several programs
  - Not a topic in this course

39.6. Attributes

Lecture 10 - slide 36

In our treatment of serialization we made extensive use of attributes, see for instance Section 39.3. In this section we will discuss attributes at a more general level, and independent of serialization.

Attributes offer a mechanism that allows the programmer to extend the programming language in simple ways. Attributes allow the programmer to associate extra information (meta data) to selected and pre-defined constructs in C#. The constructs to which it is possible to attach attributes are assemblies, classes, structs, constructors, delegates, enumeration types, fields (variables), events, methods, parameters, properties, and returns.

We all know that members of a class in C# have associated visibility modifiers, see Section 11.16. In case visibility modifiers were not part of C#, we could have used attributes as a way to extend the language with different kinds of member visibilities. Certain attributes can be accessed by the compiler, and hereby these attributes can affect the checking done by the compiler and the code generated by the compiler. Attributes can also be accessed at run-time. There are ways for the running program to access the attributes of given constructs, such that the attribute and attribute values can affect the program execution.

Program 39.7 illustrates the use of the predefined `Obsolete` attribute. Being "obsolete" means "no longer in use". In line 3, the attribute is associated with class `C`. In line 9, another usage of the attribute is associated with method `M` in class `D`.

```
1  using System;
2  
3  [Obsolete("Use class D instead")]
4  class C{
5    // ...
6  }
7  
8  class D{
9    [Obsolete("Do not call this method.",true)]
10   public void M(){
11   }
12 }
13 
14 class E{
15   public static void Main(){
16     C c = new C();
17     D d = new D();
18     d.M();
19   }
20 }
```

Program 39.7 *An obsolete class C, and a class D with an obsolete method M.*

The compiler is aware of the `Obsolete` attribute. When we compile Program 39.7 we can see the effect of the attribute, see Listing 39.8.

```
>csc prog.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

prog.cs(16,5): warning CS0618: 'C' is obsolete: 'Use class D instead'
prog.cs(16,15): warning CS0618: 'C' is obsolete: 'Use class D instead'
prog.cs(18,5): error CS0619: 'D.M()' is obsolete: 'Do not call this method.'
```

Listing 39.8 *Compiling class C, D, and E.*

C# comes with a lot of predefined attributes. `Obsolete` is one of them, and we encountered quite a few in Section 39.3 in the context of serialization. Unit testing frameworks for C# also heavily rely on attributes.

It is also possible to define our own attributes. An attribute is defined as a class. Attributes defined in this way are subclasses of the class `System.Attribute`. As a naming convention, the names of all attribute classes should have "`Attribute`" as a suffix. Thus, an attribute `X` is defined by a class `XAttribute`, which inherits from the class `System.Attribute`. The attribute usage notation **`[X(a,b,c)]`** in front of some C# construct *C* causes an instance of class `XAttribute`, made with the appropriate three-parameter constructor, to be associated with *C*. In the attribute usage notation **`[X(a,b,c,d=e)]`** `d` refers to a property of class `XAttribute`. The property `d` must be read-write (both gettable and settable), see Section 18.5. Thus, as it appears, an attribute accepts both *positional parameters* and *keyword parameters*.

Below, in Program 39.9 we have reproduced the class behind the `Obsolete` attribute. You should notice the three different constructors and the read/write property `IsError`. The attribute `AttributeUsage` attribute in 5-6 illustrates how attributes help define attributes. `AttributeUsage` define the constructs to which it possible to associate the `MyObsolete` attribute. The expression `AttributeTargets.Method | AttributeTargets.Property` denotes two values in the *combined enumeration type* `AttributeTargets` which carries a so-called flag attribute. Combined enumerations are discussed in Focus box 6.3.

```
1  // In part, reproduced from the book "C# to the Point"
2
3  using System;
4
5  [AttributeUsage(AttributeTargets.Method |
6                  AttributeTargets.Property)]
7  public sealed class MyObsoleteAttribute: Attribute{
8    string message;
9    bool isError;
10
11   public string Message{
12     get {
13       return message;
14     }
15   }
16
17   public bool IsError{
18     get {
19       return isError;
20     }
21     set {
22       isError = value;
23     }
24   }
25
26   public MyObsoleteAttribute(){
27     message = ""; isError = false;
28   }
29
```

```
  public MyObsoleteAttribute(string msg){
    message = msg; isError = false;
  }

  public MyObsoleteAttribute(string msg, bool error){
    message = msg; isError = error;
  }

}
```

Program 39.9   *A reproduction of class ObsoleteAttribute.*

In Program 39.10 we show a sample use of the attribute programmed in Program 39.9. The program does not compile because we attempt to associate the `MyObsolete` attribute to a class in line 3. As explained above, we have restricted `MyObsolete` to be connected with only methods and properties.

```
1  using System;
2  
3  [MyObsolete("Use class D instead")]
4  class C{
5    // ...
6  }
7  
8  class D{
9    [MyObsolete("Do not call this method.",IsError=true)]
10   public void M(){
11   }
12 }
13 
14 class E{
15   public static void Main(){
16     C c = new C();
17     D d = new D();
18     d.M();
19   }
20 }
```

Program 39.10   *Sample usage of the reproduced class - causes a compilation error.*

40. Patterns and Techniques

In relation to streams, which we discussed in Chapter 37 in the beginning of the IO lecture, it is relevant to bring up the ***Decorator*** design pattern. Therefore we conclude the IO lecture with a discussion of ***Decorator***.

40.1. The Decorator Pattern

Lecture 10 - slide 38

It is often necessary to extend an object of class `C` with extra capabilities. As an example, the `Draw` method of a `Triangle` class can be extended with the traditional angle and edge annotations for equally sized angles or edges. The typical way to solve the problem is to define a subclass of class `C` that extends `C` in the appropriate way. In this section we are primarily concerned with extensions of class `C` that do not affect the client interface of `C`. Therefore, the extensions we have in mind behave like specializations (see Chapter 25). The extensions we will deal with consist of adding additional code to the existing methods of `C`.

The decorator design pattern allows us to extend a class dynamically, at run-time. Extension by use of inheritance, as discussed above, is static because it takes place at compile-time. The main idea behind ***Decorator*** is a chain of objects, along the line illustrated in Figure 40.1. A message from `Client` to an instance of `ConcreteComponent` is passed through two instances of `ConcreteDecorator` by means of *delegation*. In order to arrange such delegation, a `ConcreteDecorator` and a `ConcreteComponent` should implement a common interface. This is important because a `ConcreteDecorator` is used as a stand in for a `ConcreteComponent`. This arrangement can for instance be obtained by the class hierarchy shown in Figure 40.2.

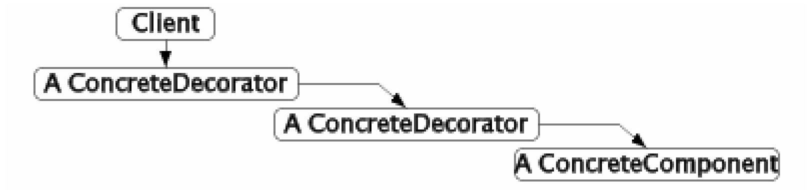

Figure 40.1 *Two decorator objects of a ConcreteComponent object*

In Figure 40.2 the `Decorators` and the `ConcreteComponent` share a common, abstract superclass called *`Component`*. When a `Client` operate on a `ConcreteComponent` it should do so via the type *`Component`*. This facilitates the object organization of Figure 40.1, because a `Decorator` can act as a stand in for a `ConcreteComponent`.

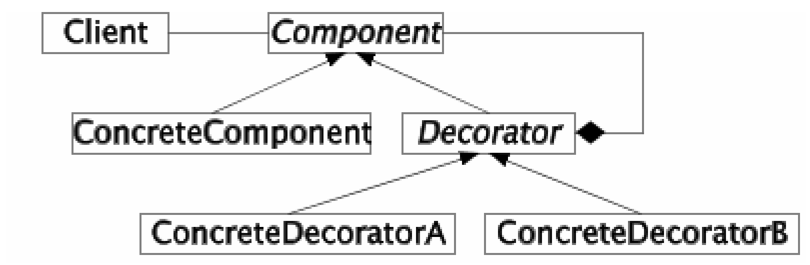

Figure 40.2 *A template of the class structure in the Decorator design pattern.*

- **Component**: Defines the common interface of participants in the Decorator pattern
- **Decorator**: References another Component to which it delegates responsibilities

The class diagram of ***Decorator*** is similar to ***Composite***, see Section 32.1. In Figure 40.2 a ***Decorator*** is intended to aggregate (reference) a single *`Component`*. In Figure 32.1 a ***Composite*** typically aggregate two or more *`Component`*s . Thus, a ***Composite*** typically gives rise to trees, whereas a ***Decorator*** gives rise to a linear lists.

`Decorator` objects can be added and chained at run-time. A `Client` accesses the outer *`Component`* (typically a `ConcreteDecorator`), which delegates part of the work to another *`Component`*. While passing, it does part of the work itself.

Use of ***Decorator*** can be seen as a dynamic alternative to static subclassing

40.2. The Decorator Pattern and Streams

Lecture 10 - slide 40

The ***Decorator*** discussion above in Section 40.1 was abstract and general. It is not obvious how it relates to streams and IO. We will now introduce the stream decorators that drive our interest in the pattern. The following summarizes the stream classes that are involved.

We build a **compressed stream** on a **buffered stream** on a **file stream**

The **compressed stream** decorates the **buffered stream**

The **buffered stream** decorates the **file stream**

The idea behind the decoration of class `FileStream` (see Section 37.4) is to supply additional properties of the stream. The additional properties in our example are *buffering* and *compression*. Buffering may result in better performance because many read and write operations do not need to touch the harddisk as such. Use of compression means that the files become smaller. (Notice that class `FileStream` already apply buffering itself, and as such the buffer decoration is more of illustrative nature than of practical value).

Figure 40.3 corresponds to Figure 40.1. Thus, Figure 40.3 shows objects, not classes. A `FileStream` object is decorated with buffering and compression. A `Client` program is able to operate on `GZipStream` (a compressed stream) as if it was a `FileStream`.

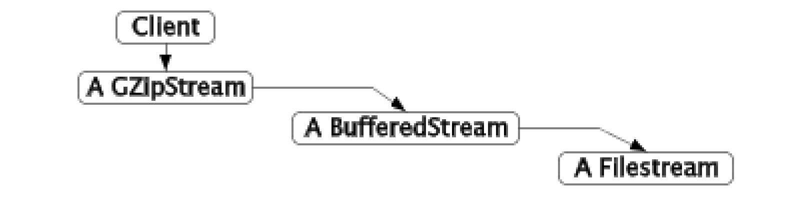

Figure 40.3 *Compression and buffering decoration of a FileStream*

In Program 40.1 we read a `FileStream` into a buffer of type `byte[]`. This is done in line 11-16. In line 18-27 we establish the decorated `FileStream` (see the **purple** parts). In line 27 we write the buffer to the decorated stream. In line 29-32 we compare the size of the original file and the compressed file. We see the effect in Listing 40.2 when the program is applied on its own source file.

```
1  using System;
2  using System.IO;
3  using System.IO.Compression;
4
5  public class CompressProg{
6
7    public static void Main(string[] args){
8      byte[] buffer;
9      long originalLength;
10
11     // Read a file, arg[0], into buffer
12     using(Stream infile = new FileStream(args[0], FileMode.Open)){
13        buffer = new byte[infile.Length];
14        infile.Read(buffer, 0, buffer.Length);
15        originalLength = infile.Length;
16     }
17
18     // Compress buffer to a GZipStream
19     Stream compressedzipStream =
20        new GZipStream(
21           new BufferedStream(
22                   new FileStream(
23                         args[1], FileMode.Create),
24                   128),
25           CompressionMode.Compress);
26     compressedzipStream.Write(buffer, 0, buffer.Length);
27     compressedzipStream.Close();
28
29     // Report compression rate:
30     Console.WriteLine("CompressionRate: {0}/{1}",
31                       MeasureFileLength(args[1]),
32                       originalLength);
33
34   }
35
36   public static long MeasureFileLength(string fileName){
37     using(Stream infile = new FileStream(fileName, FileMode.Open))
38       return infile.Length;
39   }
40
41 }
```

Program 40.1 *A program that compresses a file.*

```
1 > compress compress.cs out
2 CompressionRate: 545/1126
```

Listing 40.2 *Sample application together with program output (compression rate).*

When Program 40.1 is executed, a compressed file is written. In Program 40.3 we show how to read this file back again. In line 11-17 we set up the decorated stream, very similar to Program 40.1. In line 21-28 we read the compressed file into the buffer, and finally in line 32-35 we write the buffer back to an uncompressed file.

```
using System;
using System.IO;
using System.IO.Compression;

public class CompressProg{

  public static void Main(string[] args){
    byte[] buffer;
    const int LargeEnough = 10000;

    Stream compressedzipStream =
       new GZipStream(
         new BufferedStream(
                new FileStream(
                      args[0], FileMode.Open),
                128),
         CompressionMode.Decompress);

    buffer = new byte[LargeEnough];

    // Read and decompress the compressed stream:
    int bytesRead = 0,
        bufferPtr = 0;
    do{
      // Read chunks of 10 bytes per call of Read:
      bytesRead = compressedzipStream.Read(buffer, bufferPtr, 10);
      if (bytesRead != 0) bufferPtr += bytesRead;
    } while (bytesRead != 0);

    compressedzipStream.Close();

    // Write contens of buffer to the output file
    using(Stream outfile = new FileStream(args[1], FileMode.Create)){
       outfile.Write(buffer, 0, bufferPtr);
    }
  }

}
```

Program 40.3 *The corresponding program that decompresses the file.*

With this we are done with the IO lecture.

41. Motivation for Generic Types

This chapter starts the lecture about *generics*: Generic types and generic methods. With *generics* we are aiming at more general types (classes, structs, interfaces, etc). The measure that we will bring into use is *type parametrization*.

This chapter is intended as motivation. Type parameterized types will be the topic of Chapter 42 and type parameterized methods will be treated in Chapter 43.

41.1. Operations on sets

Lecture 11 - slide 2

In this chapter we decide to develop and use the class `Set`. We use the class `Set` as a motivating example. It is our goal, once and for all, to be able to write a class `Set` that supports all possible types of elements. It is the intention that the class `Set` can be used in any future program, in which there is a need for sets.

It is noteworthy that .NET has not supported a mathematical set class until version 3.5. As of version 3.5, the class `HashSet<T>` supports sets, see also Section 45.1. Thus, at the time of writing this material, there was no set class available in the .NET Framework.

The class `Set` should represent a mathematical set of items. We equip class `Set` with the usual and well-known set operations:

- *aSet*.**`Member`**(*element*)
- *aSet*.**`Insert`**(*element*)
- *aSet*.**`Delete`**(*element*)
- *aSet*.**`Count`**
- *aSet*.**`Subset`**(*anotherSet*)
- *aSet*.**`GetEnumerator`**()
- *aSet*.**`Intersection`**(*anotherSet*)
- *aSet*.**`Union`**(*anotherSet*)
- *aSet*.**`Diff`**(*anotherSet*)

The set operations `Intersection`, `Union`, and `Diff` are handled in Exercise 11.1.

41.2. The classes IntSet and StringSet

Lecture 11 - slide 3

Let us imagine that we first encounter a need for sets of integers. This causes us (maybe somewhat narrow-minded) to write a class called `IntSet`. Our version of class `IntSet` is shown in Program

41.1. The version provided in the paper version of the material is abbreviated to save some space. The version in the web version is complete with all details.

```
using System;
using System.Collections;

public class IntSet {

  private int capacity;
  private static int DefaultCapacity = 10;
  private int[] store;
  private int next;

  public IntSet(int capacity){
    this.capacity = capacity;
    store = new int[capacity];
    next = 0;                          // The next place to insert
  }

  public IntSet(): this(DefaultCapacity){
  }

  public IntSet(int[] elements): this(elements.Length){
    foreach(int  el in elements) this.Insert(el);
  }

  // Copy constructor
  public IntSet(IntSet s): this(s.capacity){
    foreach(int  el in s) this.Insert(el);
  }

  public bool Member(int  element){
    for(int idx = 0; idx < next; idx++)
      if (element.Equals(store[idx]))
        return true;
    return false;
  }

  public void Insert(int  element){
    if (!this.Member(element)){
      if (this.Full){
        Console.WriteLine("[Resize to {0}]", capacity * 2);
        Array.Resize<int>(ref store, capacity * 2);
        capacity = capacity * 2;
      }
      store[next] = element;
      next++;
    }
  }

  public void Delete(int  element){
    bool found = false;
    int foundIdx = 0;
    for(int idx = 0; !found && (idx < next); idx++){
      if (element.Equals(store[idx])){
         found = true;
         foundIdx = idx;
      }
    }
    if (found){   // shift remaining elements left
      for(int idx = foundIdx+1; idx < next; idx++)
```

```
        store[idx-1] = store[idx];
      store[next-1] = default(int  );
      next--;
    }
  }

  // Additional operations: Count, Subset, ToString, Full, and GetEnumerator

}
```

Program 41.1 *The class IntSet.*

The class `IntSet` is an example of an everyday implementation of integer sets. We have not attempted to come up with a clever representation that allows for fast set operations. The `IntSet` class is good enough for small sets. If you are going to work on sets with many elements, you should use a set class of better quality.

We chose to represent the elements in an integer array. We keep track of the position where to insert the next element (by use of the instance variable `next`). If there is not enough room in the array, we use the `Array.Resize` operation to make it larger. We delete elements from the set by shifting elements in the array 'to the left', in order to avoid wasted space. This approach is fairly expensive, but it is good enough for our purposes. The `IntSet` class is equipped with a `GetEnumerator` method, which returns an iterator. (We encountered iterators (enumerators) in the `Interval` class studied in Section 21.3. See also Section 31.6 for details on iterators. The `GetEnumerator` details are not shown in the paper version). The enumerator allows for traversal of all elements of the set with a **`foreach`** control structure.

A set is only, in a minimal sense, dependent on the types of elements (in our case, the type `int`). It does not even matter if the type of elements is a value type or a reference type (see Section 14.1 and Section 13.1 respectively). We do, however, apply equality on the elements, via use of the `Equals` method. Nevertheless, the type `int` occurs many times in the class definition of `IntSet`. We have emphasized occurrences of `int` with color marks in Program 41.1.

```
using System;
using System.Collections;

class App{

 public static void Main(){
   IntSet s1 = new IntSet(),
          s2 = new IntSet();

   s1.Insert(1); s1.Insert(2);  s1.Insert(3);
   s1.Insert(4); s1.Insert(5);  s1.Insert(6);
   s1.Insert(5); s1.Insert(6);  s1.Insert(8);
   s1.Delete(3); s1.Delete(6);  s1.Insert(9);

   s2.Insert(8); s2.Insert(9);

   Console.WriteLine("s1: {0}", s1);
   Console.WriteLine("s2: {0}", s2);

// Exercises:
// Console.WriteLine("{0}", s2.Intersection(s1));
22 // Console.WriteLine("{0}", s2.Union(s1));
23 // Console.WriteLine("{0}", s2.Diff(s1));
24
25    if (s1.Subset(s2))
26      Console.WriteLine("s1 is a subset of s2");
27    else
28      Console.WriteLine("s1 is not a subset of s2");
29
30    if (s2.Subset(s1))
31      Console.WriteLine("s2 is a subset of s1");
32    else
33      Console.WriteLine("s2 is not a subset of s1");
34  }
35 }
```

Program 41.2   *A client of IntSet.*

In Program 41.2 we see a sample application of class `IntSet`. We establish two empty integer sets `s1` and `s2`, we insert some numbers into these, and we try out some of the set operations on them. The comment lines 20-23 make use of set operations which will be implemented in Exercise 11.1. The output of Program 41.2 confirms that `s2` is a subset of `s1`. The program output is shown in Listing 41.3 (only on web).

We will now assume that we, a couple of days after we have programmed class `IntSet`, realize a need of class `StringSet`. Too bad! Class `StringSet` is almost like `IntSet`. But instead of occurrences of `int` we have occurrences of `string`.

We know how bad it is to copy the source text of `IntSet` to a new file called `StringSet`, and to globally replace 'int' with 'string'. When we need to modify the set class, all our modifications will have do be done twice!

For illustrative purposes - and despite the observation just described - we have made the class `StringSet`, see Program 41.4 (only on web). We have also replicated the client program, in Program 41.5 (only on web) and the program output in Listing 41.6 (only on web).

41.3. The class ObjectSet

Lecture 11 - slide 4

In Section 41.2 we learned the following lesson:

> There is an endless number of *Type*`Set` classes. One for each *Type*. Each of them is similar to the others.

We will now review the solution to the problem which was used in Java before version 1.5, and in C# before version 2. These are the versions of the two languages prior to the introduction of generics.

The idea is simple: We implement a set class of element type `Object`. We call it `ObjectSet`. The type `Object` is the most general type in the type system (see Section 28.2). All other types inherit from the class `Object`.

Below, in Program 41.7 we show the class `ObjectSet`. In the paper version, only an outline with a few constructors and methods is included. The web version shows the full definition of class `ObjectSet`.

```
using System;
using System.Collections;

public class ObjectSet {

  private int capacity;
  private static int DefaultCapacity = 10;
  private Object[] store;
  private int next;

  public ObjectSet(int capacity){
    this.capacity = capacity;
    store = new Object[capacity];
    next = 0;
  }

  // Other constructors

  public bool Member(Object  element){
    for(int idx = 0; idx < next; idx++)
      if (element.Equals(store[idx]))
        return true;
    return false;
  }

  public void Insert(Object  element){
    if (!this.Member(element)){
      if (this.Full){
        Console.WriteLine("[Resize to {0}]", capacity * 2);
        Array.Resize<Object>(ref store, capacity * 2);
        capacity = capacity * 2;
      }
      store[next] = element;
      next++;
    }
  }

  // Other methods

}
```

Program 41.7   *An outline of the class ObjectSet.*

We can now write programs with a set of `Die`, a set of `BankAccount`, a set of `int`, etc. In Program 41.8 (only on web) we show a program, similar to Program 41.2, which illustrates sets of `Die` objects. (The class `Die` can be found in Section 10.1).

The main problem with class `ObjectSet` is illustrated below in Program 41.10. In line 12-20 we make a set of dice (`s1`), a set of integers (`s2`), a set of strings (`s3`), and set of mixed objects (`s4`). Let us focus on `s1`. If we take a die out of `s1` with the purpose of using a `Die` operation on it, we need to typecase the element to a `Die`. This is shown in line 23. From the compiler's point of view, all elements in the set `s1` are instances of class `Object`. With the cast `(Die)o` in line 23, we guarantee that each element in the set is a `Die`. (If an integer or a playing card should sneak into the set, an exception will be thrown). - The output of the program is shown in Listing 41.11 (only on web).

```
1  using System;
2  using System.Collections;
3
4  class App{
5
6   public static void Main(){
7     Die d1 = new Die(6),  d2 = new Die(10),
8         d3 = new Die(16), d4 = new Die(8);
9     int sum = 0;
10    string netString = "";
11
12    ObjectSet
13      s1 = new ObjectSet(                  // A set of dice
14             new Die[]{d1, d2, d3, d4}),
15      s2 = new ObjectSet(                  // A set of ints
16             new Object[]{1, 2, 3, 4}),
17      s3 = new ObjectSet(                  // A set of strings
18             new string[]{"a", "b", "c", "d"}),
19      s4 = new ObjectSet(                  // A set of mixed things...
20             new Object[]{new Die(6), "a", 7});
21
22    foreach(Object o in s1){
23       ((Die)o).Toss();
24       Console.WriteLine("{0}", (Die)o);
25    }
26
27    // Some details have been left out
28
29  }
30 }
```

Program 41.10 *A client of ObjectSet - working with set of different types.*

41.4. Problems

Lecture 11 - slide 5

The classes `IntSet`, `StringSet` and `ObjectSet` suffer from both programming and type problems:

- Problems with **`IntSet`** and **`StringSet`**
  - Tedious to write both versions: *Copy and paste* programming.
  - Error prone to maintain both versions
- Problems with **`ObjectSet`**
  - Elements of the set must be downcasted in case we need to use some of their specialized operations
  - We can create an inhomogeneous set
    - A set of "apples" and "bananas"

Generic types, to be introduced in the following chapter, offer a type safe alternative to `ObjectSet`, in which we are able to avoid type casting.

42. Generic Types

Generic types are types that carry type parameters. Type parameterized classes will be of particular importance. The motivation for working with type parameterized classes was gained in Chapter 41.

42.1. The generic class Set<T>

Lecture 11 - slide 7

Let us, right away, present the generic set class **`Set<T>`**. It is shown in Program 42.1. As usual, we show an abbreviated version of the class in the paper edition of the material.

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

public class Set<T> {

  private int capacity;
  private static int DefaultCapacity = 10;
  private T[] store;
  private int next;

  public Set(int capacity){
    this.capacity = capacity;
    store = new T[capacity];
    next = 0;
  }

  public Set(): this(DefaultCapacity){
  }

  public Set(T[] elements): this(elements.Length){
    foreach(T el in elements) this.Insert(el);
  }

  // Copy constructor
  public Set(Set<T> s): this(s.capacity){
    foreach(T el in s) this.Insert(el);
  }

  public bool Member(T element){
    for(int idx = 0; idx < next; idx++)
      if (element.Equals(store[idx]))
        return true;
    return false;
  }

  public void Insert(T element){
    if (!this.Member(element)){
      if (this.Full){
        Console.WriteLine("[Resize to {0}]", capacity * 2);
        Array.Resize<T>(ref store, capacity * 2);
        capacity = capacity * 2;
      }
      store[next] = element;
      next++;
    }
  }

  public void Delete(T element){
    bool found = false;
    int  foundIdx = 0;
    for(int idx = 0; !found && (idx < next); idx++){
      if (element.Equals(store[idx])){
         found = true;
         foundIdx = idx;
      }
    }
    if (found){   // shift remaining elements left
      for(int idx = foundIdx+1; idx < next; idx++)
        store[idx-1] = store[idx];
      store[next-1] = default(T);
      next--;
    }
  }

  // Additional operations: Count, Subset, ToString, Full, and GetEnumerator

}
```

Program 42.1 *The class* ***Set<T>***.

The advantage of class **Set<T>** over class **ObjectSet** becomes clear when we study a client of **Set<T>**. Please take a look at Program 42.2 and compare it with Program 41.10. We are able to work with both sets of value types, such as **Set<int>**, and sets of reference types, such as **Set<Die>**. When we take an element out of the set it is not necessary to cast it, as in Program 41.10. Notice that a **foreach** loop does not provide the best illustration of this aspect, because the *type* in `foreach(`*type* `var in collection)` is used implicitly for casting a value in collection to *type*. The only way to access elements in a set is to use its iterator. Please take a look at Exercise 11.2 if you wish to go deeper into this issue.

```
using System;
using System.Collections;

class App{

 public static void Main(){
   Die d1 = new Die(6),  d2 = new Die(10),
       d3 = new Die(16), d4 = new Die(8);
   int sum = 0;
   string netString = "";


   // Working with sets of dice:
   Set<Die>  s1 = new Set<Die>(       // A set of dice
                      new Die[]{d1, d2, d3, d4});
   foreach(Die d in s1){
      d.Toss();
      Console.WriteLine("{0}", d);
   }

```

```
    // Working with sets of ints
    Set<int> s2 = new Set<int>(         // A set of ints
                       new int[]{1, 2, 3, 4});
    foreach(int i in s2)
       sum += i;
    Console.WriteLine("Sum: {0}", sum);


    // Working with sets of strings
    Set<string> s3 = new Set<string>(  // A set of strings
                       new string[]{"a", "b", "c", "d"});
    foreach(string str in s3)
       netString += str;
    Console.WriteLine("Appended string: {0}", netString);

  }
}
```

Program 42.2 *A client of* **`Set<T>`** *- working with sets of different types.*

The output of Program 42.2 is shown in Listing 42.3 (only on web).

Exercise 11.1. *Intersection, union, and difference: Operations on sets*

On the accompanying slide we have shown a generic class **`set<T>`**.

Add the classical set operations intersection, union and set difference to the generic class **`set<T>`**.

Test the new operations from a client program.

Hint: The enumerator, that comes with the class **`set<T>`**, may be useful for the implementation of the requested set operations.

Exercise 11.2. *An element access operation on sets*

The only way to get access to an element from a set is via use of the enumerator (also known as the iterator) of the set. In this exercise we wish to change that.

Invent some operation on the set that allows you to take out an existing element in the set. This corresponds to accessing a given item in an array or a list, for instance via an indexer: `arr[i]` and `lst[j]`. Notice in this context that there is no order between elements in the set. It is not natural to talk about "the first" or "the last" element in the set.

Given the invented operation in **`Set<T>`** use it to illustrate that, for some concrete type **`T`**, no casting is necessary when elements are accessed from **`Set<T>`**

42.2. Generic Types

Lecture 11 - slide 8

Let us now describe the general concepts behind Generic Types in C#. C# supports not only generic classes, but also generic structs (see Section 42.7), generic interfaces (see Section 42.8), and generic delegate types (see Section 43.2). Overall, we distinguish between templates and constructed types:

- Templates
  - **`C<T>`** is not a type
  - **`C<T>`** is a template from which a type can be constructed
  - **`T`** is a *formal type parameter*
- Constructed type
  - The type constructed from a template
  - **`C<int>`**, **`C<string>`**, and **`D<C<int>>`**
  - **`int`**, **`string`**, and **`C<int>`** are *actual type parameters* of **`C`** and **`D`**

When we talk about a generic type we do it in the meaning of a template.

The word "template" is appropriate, and in fact just to the point. But most C# writers do not use it, because the word "template" it used in C++ in a closely related, but slightly different meaning. A template in C++ is a type parameterized class, which is expanded at compile time. Each actual type parameter will create a new class, just like we would create it ourselves in a text editor. In C#, generic classes are able to share the class representation at run-time. For more details on these matters, consult for instance [Golding05].

As a possible coding style, it is often recommended to use capital, single letter names (such as `S`, `T`, and `U`) as formal type parameters. In that way it becomes easier to recognize templates, to spot formal type names in our programs, to keep templates apart from constructed types, and to avoid very name clauses of generic types. In situations where a type takes more than one formal type parameters, an alternative coding style calls for formal type parameter names like `T`*x* and `T`*y*, (such as `TKey` and `TValue`) where *x* and *y* describe the role of each of the formal type parameters.

The ability to have generic types is known as *parametric polymorphism*

42.3. Constraints on Formal Type Parameters

Lecture 11 - slide 9

Let us again consider our implementation of the generic class **`Set<T>`** in Program 42.1. Take a close look at the class, and find out if we make any assumptions about the formal type parameter **`T`** in Program 42.1. Will any type **`T`** really apply? Please consider this, before you proceed!

In **`Set<T>`** it happens to be the case that we do not make any assumption of the type parameter `T`. This is typical for *collection classes* (which are classes that serve as element *containers*).

It is possible to express a number of constraints on a formal type parameter

The more constraints on **T**, the more we can do on **T**-objects in the body of **C<T>**

Sometimes we write a parameterized class, say **C<T>**, in which we wish to be able to make some concrete assumptions about the type parameter T. You may ask what we want to express. We could, for instance, want to express that

1. **T** is a value type, allowing for instance use of the type **T?** (nullable types, see Section 14.9) inside **C<T>** .
2. **T** is a reference type, allowing, for instance, the program fragment **T v; v = null;** inside **C<T>** .
3. **T** has a multiplicative operator ***** , allowing for expressions like **T t1, t2; ... t1 * t2 ...** in **C<T>** .
4. **T** has a method named **M** , that accepts a parameter which is also of type **T** .
5. **T** has a C# indexer of two integer parameters, allowing for **T t; ... t[i, j] ...** within **C<T>** .
6. **T** is a subclass of class **BankAccount** , allowing for the program fragment **T ba; ba.AddInterests();** within **C<T>** .
7. **T** implements the interface **IEnumerable** , allowing **foreach** iterations based on T in **C<T>** , see Section 31.6 .
8. **T** is a type with a parameterless constructor, allowing the expression **new T()** in **C<T>** .

It turns out that the constraints in 1, 2, 6, 7, and 8 can be expressed directly in C#. The constraints in 4 and 5 can be expressed indirectly in C#, whereas the constraint in 3 cannot be expressed in C#.

Here follows a program fragment that illustrates the legal form of *constraints* on type parameters in generic types in C#. We define generic classes C, E, F, and G all of which are subclasses of class D. A and B are classes defined elsewhere. The constraints are colored in Program 42.4.

```
1  class C<S,T>: D
2      where T: A, ICloneable
3      where S: B {
4   ...
5  }
6  
7  class E<T>: D
8      where T: class{
9   ...
10 }
11 
12 class F<T>: D
13     where T: struct{
14  ...
15 }
16 
17 class G<T>: D
18     where T: new(){
19  ...
20 }
```

Program 42.4 *Illustrations of the various constraints on type parameters.*

The class `C` has formal type parameters `S` and `T`. The first constraint requires that `T` is `A`, or a subclass of `A`, and that it implements the interface `IClonable`. Thus, only class `A` or subclasses of `A` that implement `IClonable` can be used as actual parameter corresponding to `T`. The type parameter `S` must be `B` or a subclass of `B`.

The class `E` has a formal type parameter `T`, which must be a class. In the same way, the class `F` has a formal type parameter `T`, which must be a struct.

The class `G` has a formal type parameter `T`, which must have a parameterless constructor.

As a consequence of the inheritance rules in C#, only a single class can be given in a constraint. Multiple interfaces can be given. A class should come before any interface. Thus, in line 2 of Program 42.4, where `T` is constrained by `A, ICloneable`, `A` can be a class, and everything after `A` in the constraint need to be interfaces.

42.4. Constraints: Strings of comparable elements

Lecture 11 - slide 10

We will now program a generic class with constraints. We will make a class `String<T>` which generalizes `System.String` from C#. An instance of `String<T>` contains a sequence of `T`-values/`T`-objects. In contrast, an instance of `System.String` contains a sequence of Unicode characters. With use of `String<T>` we can for instance make a string of integers, a string of bank accounts, and a string of dice.

Old-fashioned character strings can be ordered, because we have an ordering of characters. The ordering we have in mind is sometimes called *lexicographic ordering*, because it reflects the ordering of words in dictionaries and encyclopedia. We also wish to support ordering of our new generalized strings from `String<T>`. It can only be achieved if we provide an ordering of the values/objects in `T`. This is done by requiring that `T` implements the interface `IComparable`, which has a single method `CompareTo`. For details on `IComparable` and `CompareTo`, please consult Section 31.5.

Now take a look at the definition of `String<T>` in Program 42.5. In line 3 we state that `String<T>` should implement the interface `IComparable<String<T>>`. It is important to understand that we hereby commit ourselves to implement a `CompareTo` method in `String<T>`.

You may be confused about the interface `IComparable`, as discussed in Program 42.5 in contrast to `IComparable<S>`, which is used as `IComparable<String<T>>` in line 3 of Program 42.5. `IComparable<S>` is a generic interface. It is generic because this allows us to specify the parameter to the method `CompareTo` with better precision. We discuss the generic interface `IComparable<S>` in Section 42.8.

There is an additional important detail in line 3 of Program 42.5, namely the constraint, which is colored. The constraint states that the type `T` must be `IComparable` itself (again using the generic version of the interface). In plain English it means that there must be a `CompareTo` method available on the type, which we provide as the actual type parameter of our new string class. Our plan is, of course, to use the `CompareTo` method of `T` to program the `CompareTo` method of `String<T>`.

```
1  using System;
2  
3  public class String<T>: IComparable<String<T>> where T: IComparable<T>{
4  
5    private T[] content;
6  
7    public String(){
8      content = new T[0];
9    }
10 
11   public String(T e){
12     content = new T[]{e};
13   }
14 
15   public String(T e1, T e2){
16     content = new T[]{e1, e2};
17   }
18 
19   public String(T e1, T e2, T e3){
20     content = new T[]{e1, e2, e3};
21   }
22 
23   public int CompareTo(String<T> other){
24     int thisLength = this.content.Length,
25         otherLength = other.content.Length;
26 
27     for (int i = 0; i < Math.Min(thisLength,otherLength); i++){
28       if (this.content[i].CompareTo(other.content[i]) < 0)
29          return -1;
30       else if (this.content[i].CompareTo(other.content[i]) > 0)
31         return 1;
32     }
33     // longest possible prefixes of this and other are pair-wise equal.
34     if (thisLength < otherLength)
35        return -1;
36     else if (thisLength > otherLength)
37        return 1;
38     else return 0;
39   }
40
41   public override string ToString(){
42     string res = "[";
43     for(int i = 0; i < content.Length;i++){
44       res += content[i];
45       if (i < content.Length - 1) res += ", ";
46     }
47     res += "]";
48     return res;
49   }
50
51 }
```

Program 42.5 *The generic class* ***`String<T>`***.

In line 5 we see that a string of `T`-elements is represented as an array of `T` elements. This is a natural and straightforward choice. Next we see four constructors, which allows us to make strings of zero, one, two or three parameters. This is convenient, and good enough for toy usage. For real life use, we need a general constructor that accepts an array of `T` elements. The can most conveniently be made by use of parameter arrays, see Section 20.9.

After the constructors, from line 23-39, we see our implementation of `CompareTo`. From an overall point of view we can observe that it uses `CompareTo` of type `T`, as discussed above. This is the **blue** aspects in line 28 and 30. It may be sufficient to make this observation for some readers. If you want to understand what goes on inside the method, read on.

Recall that `CompareTo` must return a negative result if the current object is less than `other`, 0 if the current object is equal to `other`, and a positive result if the current object is greater than `other`. The for-loop in line 27 traverses the overlapping prefixes of two strings. Inside the loop we return a result, if it is possible to do so. If the for-loop terminates, the longest possible prefixes of the two string are equal to each other. The lengths of the two strings are now used to determine a result.

If `T` is the type `char`, if the current string is "abcxy", and if `other` is "abcxyz", we compare "abcxy" with "abcxy" in the for loop. "abcxy" is shorter than "abcxyz", and therefore the result of the comparison -1.

The method `ToString` starting in line 41 allows us to print instances of `String<T>` in the usual way.

In Program 42.6 we see a client class of `String<T>`. We construct and compare strings of integers, strings of strings, strings of doubles, strings of booleans, and strings of dice. The dimmed method `ReportCompare` activates the `String<T>` operation `CompareTo` on pairs of such strings. `ReportCompare` is a generic method, and it will be "undimmed" and explained in Program 43.1. Take a look at the program output in Listing 42.7 and be sure that you can understand the results.

```
using System;

class StringApp{

  public static void Main(){

    ReportCompare(new String<int>(1, 2),
                  new String<int>(1));
    ReportCompare(new String<string>("1", "2", "3"),
                  new String<string>("1"));
    ReportCompare(new String<double>(0.5, 1.7, 3.0),
                  new String<double>(1.0, 1.7, 3.0));
    ReportCompare(new String<bool>(true, false),
                  new String<bool>(false, true));
    ReportCompare(new String<Die>(new Die(), new Die()),
                  new String<Die>(new Die(), new Die()));
  }

  public static void ReportCompare<T>(String<T> s, String<T> t)
    where T: IComparable<T>{
    Console.WriteLine("Result of comparing {0} and {1}: {2}",
                      s, t, s.CompareTo(t));
  }

}
```

Program 42.6 *Illustrating Strings of different types.*

```
Result of comparing [1, 2] and [1]: 1
Result of comparing [1, 2, 3] and [1]: 1
Result of comparing [0,5, 1,7, 3] and [1, 1,7, 3]: -1
Result of comparing [True, False] and [False, True]: 1
Result of comparing [[3], [6]] and [[3], [5]]: 1
```

Listing 42.7 *Output from the String of different types program.*

Exercise 11.3. *Comparable Pairs*

This exercise is inspired by an example in the book by Hansen and Sestoft: *C# Precisely*.

Program a class **`ComparablePair<T,U>`** which implements the interface **`IComparable<ComparablePair<T,U>>`**. If you prefer, you can build the class **`ComparablePair<T,U>`** on top of class **`Pair<S,T>`** from an earlier exercise in this lecture.

It is required that **`T`** and **`U`** are types that implement **`Icomparable<T>`** and **`Icomparable<U>`** respectively. How is that expressed in the class **`ComparablePair<T,U>`**?

The generic class **`ComparablePair<T,U>`** should represent a pair *(t,u)* of values/objects where *t* is of type `T` and *u* is of type `U`. The generic class should have an appropriate constructor that initializes both parts of the pair. In addition, there should be properties that return each of the parts. Finally, the class should - of course - implement the operation `CompareTo` because it is prescribed by the interface **`System.IComparable<ComparablePair<T,U>>`**.

Given two pairs p = (a,b) and q= (c,d). p is considered less than q if a is less than c. If a is equal to c then b and d controls the ordering. This is similar to lexicographic ordering on strings.

If needed, you may get useful inspiration from the `Icomparable` class **`String<T>`** on the accompanying slide.

Be sure to test-drive your solution!

42.5. Another example of constraints

Lecture 11 - slide 11

We will now illustrate the need for the class and struct constraints. We have already touched on these constraints in our discussion of Program 42.4.

In Program 42.8 we have two generic classes `C` and `D`. Each of them have a single type parameter, `T` and `U` respectively. As shown with **red** color in line 7 and 15, the compiler complains. In line 7 we assign the value `null` to the variable `f` of type `T`. In line 15 we make a nullable type `U?` from `U`. (If you wish to be reminded about nullable types, consult Section 14.9). Before you go on, attempt to explain the error messages, which are shown as comments in Program 42.8.

```
1  /* Example from Hansen and Sestoft: C# Precisely */
2  
3  class C<T>{
4    // Compiler Error message:
5    // Cannot convert null to type parameter 'T' because it could
6    // be a value type. Consider using 'default(T)' instead.
7    T f = null;
8  }
9  
10 class D<U>{
11   // Compiler Error message:
12   // The type 'U' must be a non-nullable value type in order to use
13   // it as parameter 'T' in the generic type or method
14   // 'System.Nullable<T>'
15   U? f;
16 }
```

Program 42.8 *Two generic classes C and D - with compiler errors.*

In Program 42.9 we show new versions of `C<T>` and `D<U>`. Shown in **purple** we emphasize the constraints that are necessary for solving the problems.

The instance variable `f` of type `T` in `C<T>` is assigned to `null`. This only makes sense if `T` is a reference type. Therefore the **class** constraint on `T` is necessary.

The use of `U?` in `D<U>` only makes sense if `U` is a value type. (To understand this, you are referred to the discussion in Section 14.9). Value types in C# are provided by structs (see Section 6.6). The **struct** constraint on `U` is therefore the one to use.

```
1  /* Example from Hansen and Sestoft: C# Precisely */
2  
3  class C<T> where T: class{
4    T f = null;
5  }
6  
7  class D<U> where U: struct{
8    U? f;
9  }
10 
11 class Appl{
12 
13   // Does NOT compile:
14   C<double> c = new C<double>();
15   D<A>      d = new D<A>();
16 
17   // OK:
18   C<A>      c1 = new C<A>();
19   D<double> d1 = new D<double>();
20 
21 }
22 
23 class A{}
```

Program 42.9 *Two generic classes C and D - with the necessary constraints.*

In line 11-21 we show clients of `C<T>` and `D<U>`. The compiler errors in line 14 and 15 are easy to explain. The type `double` is not a reference type, and `A`, which is programmed in line 23, is not a value type. Therefore `double` and `A` violate the constraints of `C<T>` and `D<U>`. In line 18 and 19 we switch the roles of `double` and `A`. Now everything is fine.

42.6. Variance

Lecture 11 - slide 12

Consider the question asked in the following box.

A **`CheckAccount`** ***is a*** **`BankAccount`**

But is a **`Set<CheckAccount>`** a **`Set<BankAccount>`** ?

You are encouraged to review our discussion of the ***is a*** relation in Section 25.2.

The question is how `Set<T>` is varies when `T` varies. Variation in this context is specialization, cf. Chapter 25. Is `Set<T>` specialized when `T` is specialized?

Take a look at Program 42.10. In line 7-14 we construct a number of bank accounts and check accounts, and we make a set of bank accounts (`s1`, in line 17) and a set of check accounts (`s2`, in line 18). In line 21 and 22 we populate the two sets. So far so good. Next, in line 25 (shown in **purple**) we play the polymorphism game as we have done many times earlier, for example in line

13 of Program 28.17. If `Set<CheckAccount>` ***is a*** `Set<BankAccount>` line 25 of Program 42.10 should be OK (just as line 13 of Program 28.17 is OK).

The compiler does not like line 25, however. The reason is that `Set<CheckAccount>` ***is NOT a*** `Set<BankAccount>`.

If we for a moment assume that `Set<CheckAccount>` ***is a*** `Set<BankAccount>` the rest of the program reveals the troubles. We insert a new `BankAccount` object in `s1`, and via the alias established in line 25, the new `BankAccount` object is also inserted into `s2`. When we in line 34-35 iterate through all the `CheckAccount` objects of the set `s2`, we encounter an instance of `BankAccount`. We cannot carry out a `SomeCheckAccountOperation` on an instance of `BankAccount`.

```
1  using System;
2  
3  class SetOfAccounts{
4  
5    public static void Main(){
6  
7      // Construct accounts:
8      BankAccount ba1 = new BankAccount("John", 0.02),
9                  ba2 = new BankAccount("Anne", 0.02),
10                 ba3 = new BankAccount("Frank", 0.02);
11 
12     CheckAccount ca1 = new CheckAccount("Mike", 0.03),
13                  ca2 = new CheckAccount("Lene", 0.03),
14                  ca3 = new CheckAccount("Joan", 0.03);
15 
16     // Constructs empty sets of accounts:
17     Set<BankAccount> s1 = new Set<BankAccount>();
18     Set<CheckAccount> s2 = new Set<CheckAccount>();
19 
20     // Insert elements in the sets:
21     s1.Insert(ba1);  s1.Insert(ba2);
22     s2.Insert(ca1);  s2.Insert(ca2);
23 
24     // Establish s1 as an alias to s2
25     s1 = s2;   // Compile-time error:
26                // Cannot implicitly convert type 'Set<CheckAccount>'
27                // to 'Set<BankAccount>'
28 
29     // Insert a BankAccount object into s1,
30     // and via the alias also in s2
31     s1.Insert(new BankAccount("Bodil", 0.02));
32 
```

```
    // Activates some CheckAccount operation on a BankAccount object
    foreach(CheckAccount ca in s2)
      ca.SomeCheckAccountOperation();

    Console.WriteLine("Set of BankAccount: {0}", s1);
    Console.WriteLine("Set of CheckAccount: {0}", s2);


  }

}
```

Program 42.10 *Sets of check accounts and bank accounts.*

The experimental insight obtained above is - perhaps - against our intuition. It can be argued that an instance of `Set<CheckAccount>` should be a valid stand in for an instance of `Set<BankAccount>`, as attempted in line 25. On the other hand, it can be asked if the extension of `Set<CheckAccount>` is a subset of `Set<BankAccount>`. (See Section 25.2 for a definition of extension). Or asked in this way: Is the set of set of check accounts a subset of a set of set of bank accounts? As designed in Section 25.3 the set of *CheckAccounts* is a subset of the set of *BankAccount*. But this does not imply that the set of set of `CheckAccount` is a subset of the set of set of `BankAccount`. A set of `CheckAccount` (understood as a single objects) is incompatible with a set of `BankAccount` (understood as a single object).

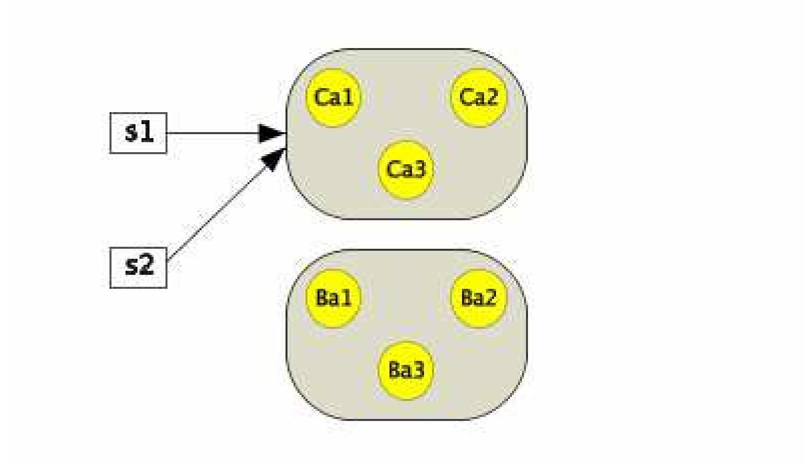

Figure 42.1 *A set of bank accounts and a set of check accounts*

In Program 42.10 we establish the scene illustrated in Figure 42.1. More precisely, the illustration shows the situation as of line 28 of Program 42.10. The problem is that we in line 31 add a new instance of `BankAccount` to `s1`, which refers to an instance of `Set<CheckAccount>`. Later in the program (line 35) this would cause "a minor explosion" if the program was allowed to reach this point . Thus, the real problem occurs if we mutate the set of check accounts that are referred from a variable of static type `Set<BankAccount>`. (See Section 28.10 for the definition of *static type*).

In general, we distinguish between the following kinds of variances in between `Set<T>` and `T`:

- **Covariance**
  - The set types vary in the same way as the element types
- **Contravariance**
  - The set types vary in the opposite way as the element types
- **Invariance**
  - The set types are not affected by the variations of the element types

If Program 42.10 could be compiled and executed without problems (if line 25 is considered OK), then we would have covariance between `Set<T>` and `T`

In C# `Set<T>` is invariant in relation to `T`.

We notice that the problem discussed above is similar to the parameter variance problem, which we discussed in Section 29.2.

C# and Java do both agree on invariance in between `Set<T>` and `T`. But in contrast to C#, Java has a solution to the problem in terms of *wildcard types*. We realized above that `Set<T>` is not a generalization of all sets. In Java 1.5, a wildcard type written as `Set<?>` (a set of unknown) is a generalization of all sets. It is, however, not possible to mutate an object of static type `Set<?>`. If you are interested to known more about generics in Java, you should consult Gilad Bracha's tutorial "Generics in the Java Programming Language", [Bracha2004].

42.7. Generic structs

Lecture 11 - slide 13

It is possible to make type parameterized structs, similar to the type parameterized classes that we have seen in the previous sections.

As an example we will see how we can define the generic struct `Nullable<T>` which defines the type behind the notation `T?` for an arbitrary value type `T`. Nullable types were discussed earlier in Section 14.9. Recall that nullable types enjoy particular compiler support, beyond the translation of `T?` to `Nullable<T>`. This includes support of lifted operators (operators that are extended to work on `T?` in addition to `T`) and support of the `null` value as such.

```
using System;

public struct Nullable<T>
  where T : struct{

  private T value;
  private bool hasValue;

  public Nullable(T value){
    this.value = value;
    this.hasValue = true;
  }

  public bool HasValue{
    get{
      return hasValue;
    }
  }

  public T Value{
    get{
      if(hasValue)
        return value;
      else throw new InvalidOperationException();
    }
  }

}
```

Program 42.11 *A partial reproduction of struct `Nullable<T>`.*

The generic struct `Nullable<T>` aggregates a value of type T and a boolean value. The boolean value is stored in the boolean instance variable `hasValue`. If `nv` is of type `Nullable<T>` for some value type `T`, and if the variable `hasValue` of `nv` is `false`, then `nv` is considered to have the value `null`. The compiler arranges that the assignment `nv = null` is translated to `nv.hasValue = false`. This is somehow done behind the scene because `hasValue` is private.

42.8. Generic interfaces: IComparable<T>

Lecture 11 - slide 14

In this section we will take a look at the generic interface `IComparable<T>`. We have earlier in the material (Section 31.5) studied the non-generic interface `Icomparable`, see Program 31.6.

If you review your solution to Exercise 8.6 you should be able to spot the weakness of a class `ComparableDie`, which implements `IComparable`. The weakness is that the parameter of the method `CompareTo` must have an `Object` as parameter. A method with the signature `CompareTo(Die)` does not implement the interface `IComparable`. (Due to static overloading of methods in C#, the methods `CompareTo(Object)` and `CompareTo(Die)` are two different methods, which just as well could have the signatures `ObjectCompareTo(Object)` and `DieCompareTo(Die)`). Thus, as given by the signature of `CompareTo`, we compare a `Die` and any possible object.

In Program 42.12 we reproduce `IComparable<T>`. Program 42.12 corresponds to Program 31.6. (Do not use any of these - both interfaces are parts of the `System` namespace). As it appears, in the generic interface the parameter of `CompareTo` is of type `T`. This alleviates the problem of the non-generic interface `IComparable`.

```
using System;

public interface IComparable <T>{
  int CompareTo(T other);
}
```

Program 42.12 *A reproduction of the generic interface* ***IComparable<T>***.

Below we show a version of class `Die` which implements the interface `IComparable<Die>`. You should notice that this allows us to use `Die` as formal parameter of the method `CompareTo`.

```
using System;

public class Die: IComparable<Die> {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private const int maxNumberOfEyes = 6;

  public Die(){
    randomNumberSupplier = Random.Instance();
    numberOfEyes = NewTossHowManyEyes();
  }

  public int CompareTo(Die other){
    return this.numberOfEyes.CompareTo(other.numberOfEyes);
  }

  // Other Die methods

}
```

Program 42.13 *A class Die that implements* ***IComparable<T>***.

The implementation of the generic interface is more type safe and less clumsy than the implementation of the non-generic solution

42.9. Generic equality interfaces

Lecture 11 - slide 15

Before reading this section you may want to remind yourself about the fundamental equality operations in C#, see Section 13.5.

There exist a couple of generic interfaces which prescribes `Equals` operations. The most fundamental is `IEquatable<T>`, which prescribes a single `Equals` instance method. It may be attractive to implement `IEquatable` in certain structs, because it could avoid the need of boxing the struct value in order to make use of the inherited `Equals` method from class `Object`.

`IEqualityComparer<T>` is similar, but it also supports a `GetHasCode` method. (Notice also that the signatures of the `Equals` methods are different in the two interfaces. `IEquatable<T>` prescribes `x.Equals(y)` whereas `IEqualityComparer<T>` prescribes `Equals(x,y)`).

Below, in Program 42.14 and Program 42.15 we show reproductions of the two interfaces. Notice again that the two interfaces are present in the namespaces `System` and `System.Collections.Generic` respectively. Use them from there if you need them.

```
using System;

public interface IEquatable <T>{
  bool Equals (T other);
}
```

Program 42.14 *A reproduction of the generic interface* ***`IEquatable<T>`***.

```
using System;

public interface IEqualityComparer <T>{
  bool Equals (T x, T y);
  int GetHashCode (T x);
}
```

Program 42.15 *A reproduction of the generic interface* ***`IEqualityComparer<T>`***.

Several operations in generic collections, such as in `List<T>` in Section 45.9, need equality operations. The `IndexOf` method in `List<T>` is a concrete example, see Section 45.11. Using `lst.IndexOf(el)` we search for the element `el` in the list `lst`. Comparison of `el` with the elements of the list is done by the *default equality comparer* of the type `T`. The abstract generic class `EqualityComparer<T>` offers a static `Default` property. The `Default` property delivers the default equality comparer for type `T`. The abstract, generic class `EqualityComparer<T>` implements the interface `IEqualityComparer<T>`.

Unfortunately the relations between the generic interfaces `IEquatable<T>` and `IEqualityComparer<T>`, the class `EqualityComparer<T>` and its subclasses are quite complicated. It seems to be the cases that these interfaces and classes have been patched several times, during the evolution of versions of the .Net libraries. The final landscape of types is therefore more complicated than it could have been desired.

42.10. Generic Classes and Inheritance

Lecture 11 - slide 16

In this section we will clarify inheritance relative to generic classes. We will answer the following questions:

Can a generic/non-generic class inherit from a non-generic/generic class?

The legal and illegal subclassings are summarized below:

- Legal subclassing
    - A generic subclass of a non-generic superclass
    - A generic subclass of a constructed superclass
    - A generic subclass of generic superclass
- Illegal subclassing
    - A non-generic subclass of generic superclass

You can refresh the terminology (generic class/constructed class) in Section 42.2.

The rules are exemplified below.

```
1  using System;
2  
3  // A generic subclass of a non-generic superclass.
4  class SomeGenericSet1<T>: IntSet{
5    // ...
6  }
7  
8  // A generic subclass of a constructed superclass
9  class SomeGenericSet2<T>: Set<int>{
10   // ...
11 }
12 
13 // A generic subclass of a generic superclass
14 // The most realistic case
15 class SpecializedSet<T>: Set<T>{
16   // ...
17 }
18 
19 // A non-generic subclass of a generic superclass
20 // Illegal. Compile-time error:
21 // The type or namespace name 'T' could not be found
22 class Set: Set<T>{
23   // ...
24 }
```

Program 42.16 *Possible and impossible subclasses of* **`Set`** *classes.*

From line 4 to 6 we are about to program a generic class `SomeGenericSet1<T>` based on a non-generic class `IntSet`. This particular task seems to be a difficult endeavor, but it is legal - in general - to use a non-generic class as a subclass of generic class.

Next, from line 9 to 11, we are about to program a generic class `SomeGenericSet2<T>` based on a constructed class `Set<int>`. This is also OK.

From line 15-17 we show the most realistic case. Here we program a generic class based on another generic class. In the specific example, we are about to specialize `Set<T>` to `SpecializedSet<T>`. The type parameter `T` of `SpecializedSet<T>` also becomes the type parameter of `Set<T>`. In general, it would also be allowed for `SpecializedSet<T>` to introduce additional type parameters, such as in `SpecializedSet<T,S> : Set<T>`.

The case shown from line 22 to 24 is illegal, simply because `T` is not the name of any known type. In line 22, `T` is name of an *actual* type parameter, but `T` is not around! It is most likely that the programmer is confused about the roles of *formal* and *actual* type parameters, see Section 42.2.

42.11. References

[Bracha2004]	Gilad Bracha, "Generics in the Java Programming Language", July 2004.
[Golding05]	Tod Golding, *Professional .NET 2.0 Generics*. Wiley Publishing, Inc., 2005.

43. Generic Methods

We are used to working with procedures, functions, and methods with parameters. Procedures, functions and methods are all known as abstractions. A parameter is like a variable that generalizes the abstraction. Each parameter of a procedure, a function, or a method is of a particular type. In this chapter we shall see how such types themselves can be passed as parameters to methods. When methods are parameterized with types, we talk about *generic methods*.

43.1. Generic Methods

Lecture 11 - slide 18

In Section 42.2 we realized that a generic type (such as a generic class) is a template from which it is possible to construct a real class. In the same way, a generic method is template from which we can construct a real method.

In C# and similar languages, all methods belong to classes. Some of these classes are generic, some are just simple, ordinary classes. We can have generic methods in both generic types, and in non-generic types.

Our first example in Program 43.1 is the generic method `ReportCompare` in the non-generic class `StringApp`. `ReportCompare` is a method in the client class of `String<T>` which we encountered in Section 42.4. When we first met it, we where not interested in the details of it, so therefore it was dimmed in Program 42.6.

Notice first that the method `ReportCompare` takes two ordinary parameters `s` and `t`. They are both of type `String<T>` for some given type `T`. The method is supposed to report the ordering of `s` relative to `t` via output written to the console. `T` is a (formal) type parameter of the method. Type parameters of methods are given in "triangular brackets" **<...>** in between the method name and the ordinary parameter list. It is highlighted with **purple** in Program 43.1.

The formal type parameter of `ReportCompare` is passed on as an actual type parameter to our generic class `String<T>` from Section 42.4. If we look at our definition of the generic class `String<T>` in Program 42.5 we notice that `T` must implement `Icomparable<T>`. This is a constraint of `T`, identical to one of the constraints of type parameters of types, see Section 42.3. The only way to ensure this in Program 43.1 is to add the constraint to the generic method. This is the **blue** part, see line 15.

Notice in line 7-11 of Program 43.1 that the actual type parameter of `ReportCompare` is not given explicitly. The actual type parameters of the five calls are conveniently inferred from the context. It is, however, possible to pass the actual type parameter explicitly. If we chose to do so, line 7 of Program 43.1 would be

```
  ReportCompare<int>(new String<int>(), new String<int>(1));
```

The remaining aspects of `ReportMethod` are simple and straightforward.

```
using System;

class StringApp{

  public static void Main(){

    ReportCompare(new String<int>(), new String<int>(1));
    ReportCompare(new String<int>(1), new String<int>(1));
    ReportCompare(new String<int>(1,2,3), new String<int>(1));
    ReportCompare(new String<int>(1), new String<int>(1,2,3));
    ReportCompare(new String<int>(1,2,3), new String<int>(1,2,3));
  }

  public static void ReportCompare<T>(String<T> s, String<T> t)
    where T: IComparable<T>{
    Console.WriteLine("Result of comparing {0} and {1}: {2}",
                      s, t, s.CompareTo(t));
  }

}
```

Program 43.1 *The generic method ReportCompare in the generic String programs.*

Let us now study an additional program example with generic methods. Program 43.2 contains a bubblesort method in line 5-11. `Bubblesort` sorts an array of element type `T`, where `T` is a type parameter of the method. The type parameter makes our bubblesort method more general, because it allow us to sort an array of arbitrary type `T`. The only requirement is, quite naturally, that objects/values of type type `T` should be comparable, such that we can ask if one value is less than or equal to another value. This is expressed by the `Icomparable<T>` constraint on T at the end of line 5.

The implementation of bubblesort in Program 43.2 has no surprises. In a double for loop we compare and swap elements. Comparison is made possible because `a[i]` values are of type `T` that implements `Icomparable<T>`. Swapping of elements are done by the `Swap` method via use of C# **ref** parameters, see Section 20.6. Notice that `Swap` is also a generic method, because it can swap values/objects of arbitrary types. Be sure to notice the formal type parameter `T` of `Swap` in line 13.

Finally we have the generic method `ReportArray`, (see line 18-21), which simply prints the values of the array to standard output.

```
1  using System;
2
3  class SortDemo{
4
5    static void BubbleSort<T>(T[] a) where T: IComparable<T>{
6     int n = a.Length;
7     for (int i = 0; i < n - 1; ++i)
8       for (int j = n - 1; j > i; --j)
9         if (a[j-1].CompareTo(a[j]) > 0)
10           Swap(ref a[j-1], ref a[j]);
11   }
12
13   public static void Swap<T>(ref T a, ref T b){
14     T temp;
15     temp = a; a = b; b = temp;
16   }
17
18   public static void ReportArray<T>(T[] a){
19     foreach(T t in a) Console.Write("{0,4}", t);
20     Console.WriteLine();
21   }
22
23   public static void Main(){
24     double[] da = new double[]{5.7, 3.0, 6.9, -5,3, 0.3};
25
26     Die[] dia = new Die[]{new Die(), new Die(),  new Die(),
27                           new Die(),  new Die(),  new Die()};
28
29     ReportArray(da); BubbleSort(da); ReportArray(da);
30     Console.WriteLine();
31     ReportArray(dia); BubbleSort(dia); ReportArray(dia);
32     Console.WriteLine();
33
34     // Equivalent:
35     ReportArray(da); BubbleSort<double>(da); ReportArray(da);
36     Console.WriteLine();
37     ReportArray(dia); BubbleSort<Die>(dia); ReportArray(dia);
38   }
39
40 }
```

Program 43.2 *A generic bubble sort program.*

In the `Main` method we make an array of doubles and an array of dice. Values of type `double` are comparable. We compile the program with a version of class `Die` that implements `IComparable<T>`, such as the `Die` class of Program 42.13. The calls of `BubbleSort` in line 29 and 31 do not supply an actual type parameter to `BubbleSort<T>`. The compiler is smart enough to infer the actual type parameter from the declared types of the variables `da` and `dia` respectively. In line 35 and 37 we show equivalent calls of `BubbleSort` to which we explicitly supply the actual type parameters `double` and `Die`.

The output of Program 43.2 is shown in Listing 43.3 (only on web).

43.2. Generic Delegates

Lecture 11 - slide 19

Delegates were introduced in Section 22.1. Recall from there that a delegate is a type of methods. In the previous section we learned about generic methods. It therefore not surprising that we also need to discuss generic delegates.

In Program 22.3 we introduced a delegate `NumericFunction`, which covers all function from `double` to `double`. In the same program we also introduced `Compose`, which composes two numeric functions to a single numeric function. In mathematical notation, the composition of *f* and *g* is denoted $f \circ g$, and it maps x to $f(g(x))$. We are now going to generalize the function `Compose`, such that it can be used on other functions of more general signatures.

Let us assume that we work with two functions *f* and *g* of the following signatures:

- $g : T \rightarrow U$
- $f : U \rightarrow S$

Thus, *g* maps a value of type T to a value of type U. *f* maps a value of type U to a value of type S. The composite function $f \circ g$ therefore maps a value of type T to a value of type S via a value of type U:

- $f \circ g : T \rightarrow S$

In line 6 of Program 43.4 we show a delegate called `Function`, which is a function type that maps a value of type S to values of type T. (It corresponds to `NumericFunction` in Program 22.3). In line 10-13 of Program 43.4 we show the function `Compose`, which we motivated above. `Function` is a *generic delegate* because it is type parameterized. `Compose` is a generic method, as discussed in Section 43.1. The generic method `PrintTableOfFunction`, shown in line 16-23, takes a `Function` `f` and an array `inputValues` of type `S[]`, and it applies and prints `f(s)` on each element `s` of `inputValues`.

```
1  using System;
2  
3  public class CompositionDemo {
4  
5    // A function from S to T
6    public delegate T Function <S,T>(S d);
7  
8    // The generic function for function composition
9    // from T to S via U
10   public static Function<T,S> Compose<T,U,S>
11                    (Function<U,S> f, Function<T,U> g){
12     return delegate(T d){return f(g(d));};
13   }
14 
15   // A generic PrintTable function
16   public static void PrintTableOfFunction<S,T>
17                    (Function<S,T> f, string fname,
18                     S[] inputValues){
19     foreach(S s in inputValues)
20       Console.WriteLine("{0,35}({1,-4:F3}) = {2}", fname, s, f(s));
21 
22     Console.WriteLine();
23   }
24 
25   // DieFromInt: int -> Die
26   public static Die DieFromInt(int i){
27     return new Die(i);
28   }
29 
30   // Round: double -> int
31   public static int Round(double d){
32     return (int)(Math.Round(d));
33   }
34 
35   public static void Main(){
36     double[] input = new double[25];
37     for(int i = 0; i < 25; i++)
38       input[i] = (double) (i*2);
39 
40     // Compose(DieFromInt, Round): double -> Die
41     // (via int)
42 
43     PrintTableOfFunction(Compose<double,int,Die>(DieFromInt, Round),
44                          "Die of double",
45                          input);
46   }
47 
48 }
```

Program 43.4 *An example that involves other types than double.*

In line 43 of `Main` we compose the two functions `DieFromInt` and `Round`. They are both programmed explicitly, in line 26 and 31 respectively. The function `Round` maps a `double` to an `int`. The function `DieFromInt` maps an `int` to a `Die`. Thus, `Compose(DieFromInt, Round)` maps a `double` to a `Die`. Notice how we pass the three involved types `double`, `int`, and `Die` as actual type parameters to `Compose` in line 43.

The version of class `Die` used in Program 43.4 can, for instance, be the class shown in Program 12.6. The parameter of the constructor determines the maximum number of eyes of the die.

The output of Program 43.4 is shown in Listing 43.5 (only on web).

43.3. Generic types and methods - Pros and Cons

Lecture 11 - slide 21

In this final section about generic types and methods we will briefly summarize the advantages and disadvantages of generics.

- Advantages
    - Readability and Documentation
        - More precise indication of types.
        - Less downcasting from class Object
    - Type Checking
        - Better and more precise typechecking
    - Efficiency
        - There is a potential for more efficient programs
        - Less casting - fewer boxings
- Disadvantages
    - Complexity
        - Yet another abstraction and parametrization-level on top of the existing

This ends the general discussion of generics. In the lecture about collections, from Chapter 44 to Chapter 48, we will make heavy use of generic types.

44. Collections - History and Overview

This chapter is the first in our coverage of collections.

Collections are used to organize and process a number of objects or values of the same type. In almost any real-life program, collections of objects or values play important roles.

Collections fit nicely in our agenda of object-oriented programming. A collection holds a number of objects (of the same type), but a concrete collection is also itself an object. The commonalities of a number of collections objects are described by the type of the collection objects. In the following chapters we will encounter a number of different interfaces and classes, which represent collection types. Not surprisingly, generic types as discussed in Chapter 42, play an important role when we wish to deal with collections that are constrained to contain only objects of a particular element type.

In the rest of this short introductory chapter we will briefly outline the historic development of collection programming. In the main part of the lecture, Chapter 45 and Chapter 46, we deal with two main categories of collections: Lists and Dictionaries.

44.1. A historic View on Collection Programming

Lecture 12 - slide 2

We identify three stages or epochs related to the development of collections:

- Native arrays and custom made lists
  - *Fixed sized arrays* - limited set of operations
  - *Variable sized linked lists* - direct pointer manipulation
- First generation collection classes
  - Elements of type `Object` - Flexible sizing - Rich repertoire of operations
  - Type unsafe - Casting - Inhomogeneous collections
- Second generation collection classes
  - The flexibility of the first generation collections remains
  - Type safe - Generic - Type parameterized - Homogeneous

Arrays are fundamental in imperative programming, for instance in C. In older programs - or old-fashioned programs - many collections are dealt with by means of arrays. Many modern programs still use arrays for collections, either due to old habits or because of the inherent efficiency of array processing. The efficiency of arrays stems from the fact that the memory needed for the elements is allocated as a single consecutive area of fixed size.

Another fundamental technique for dealing with collections is encountered in linked lists. In linked list one elements is connected to the next element by a pointer. The linking is done by use of pointers. In single-linked list, an element is linked to its successor. In double-linked list, an element is both linked to its successor and to its predecessor. Linked trees, such as binary trees, are also common. In some languages (such as C and Pascal) linked data structures require explicit pointer manipulation. Other languages (such as Lisp) hide the pointers behind the scene.

First generation collection classes deemphasize the concrete representation of collections. Instead, the capabilities and interfaces (such as insertion, deletion, searching, conversion, etc) of collections are brought into focus. This reflects good and solid object-oriented thinking. Typical first-generation collection classes blur the distinction between (consecutive) arrays and (linked) lists. The concept of an `ArrayList` is seen both in early versions of Java and C#. Collection concepts are organized in type hierarchies: A `List` ***is a*** `Collection` and a `Set` ***is a*** `Collection` (see Section 25.2). The element type of collections is the most general type in the system, namely `Object`. As a consequence of this, it is hard to avoid collection of "pears" and "bananas" (inhomogeneous collections). Thus, type safeness must be dealt with at run-time. This is against the trend of static type checking and type safety. We will briefly review the first generation collection classes of C# in Chapter 47.

The second (and current) generation of collections make use of generic types (type parameterized classes and interfaces), as discussed in Chapter 42. The weaknesses of the first generation collection classes have been the primary motivation for introduction all the complexity of genericity (see Chapter 41 where we motivated generic classes by a study of the class `Set`). With use of type parameterized classes we can statically express **`List<Banana>`** and **`List<Pear>`** and hereby eliminate the risk of type errors at run time. In the following chapters we will - with the exception of Chapter 47 - limit ourselves to study type parameterized collections.

45. Generic Collections in C#

In this chapter we will study different list interfaces and classes.

45.1. Overview of Generic Collections in C#

Lecture 12 - slide 4

We start by showing a type hierarchy of list-related types. The white boxes in Figure 45.1 are interfaces and the grey boxes are classes.

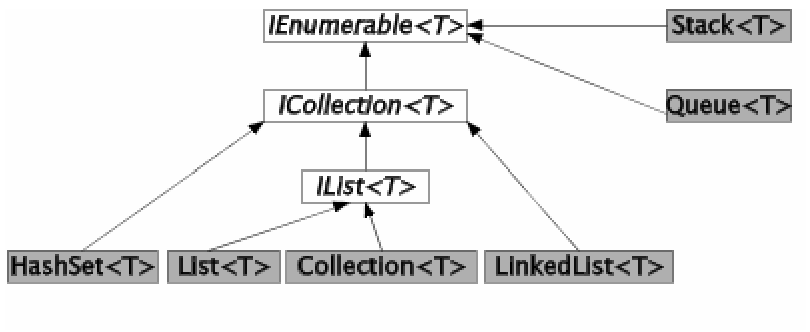

Figure 45.1 *The class and interface inheritance tree related to Lists*

All interfaces and classes seen in Figure 45.1, apart from **`Stack<T>`** and **`Queue<T>`**, will be discussed in the forthcoming sections of the current chapter.

The class `System.Array` (see Section 28.2) which conceptually is the superclass of all native array types in C#, also implements the generic interfaces `IList<T>`. Notice, however, that `Array` 's implementation of `IList<T>` is carried out by special means, and that it does not show up in the usual C# documentation. A more detailed discussion of the `Array` class is carried out in Section 47.1.

Version 3.5 of the .NET Framework contains a class, `HashSet<T>`, that supports the mathematical set concept. As such, it is similar to the class `Set<T>`, which we used as example for introduction of generic types in Section 42.1. `HashSet<T>` is, however, much more efficient than `Set<T>`.

45.2. The Interface IEnumerable<T>

Lecture 12 - slide 5

At the most general level of Figure 45.1 *traversability* is emphasized. This covers the ability to step through all elements of a collection. The interface `IEnumerable<T>` announces one parameterless method called `GetEnumerator`. The type parameter `T` is the type of the elements in the collection.

- Operations in the interface **`IEnumerable<T>`**:
    - `IEnumerator<T>` **`GetEnumerator`** ( )

As the name indicates, `GetEnumerator` returns an enumerator, which offers the following interface:

- Operations in the interface **`IEnumerator<T>`**:
  - `T` **`Current`**
  - `bool` **`MoveNext`**`( )`
  - `void` **`Reset`** `( )`

We have discussed the non-generic versions of both interfaces in Section 31.6. An `IEnumerator` object is used as the basis of traversal in a **`foreach`** loop.

Without access to an `IEnumerator` object it would not be possible to traverse the elements of a collection in a **foreach** loop. You do not very often use the `GetEnumerator` operation explicitly in your own program, but you most probably rely on it implicitly! The reason is that many of your collections are traversed, from one end to the other, by use of **foreach**. The **foreach** control structure would not work without the operation `GetEnumerator`. As you can see from Figure 45.1 all of our collections implement the interface `IEnumerable<T>` and hereby they provide the operation `GetEnumerator`.

It is worth noticing that an object of type `IEnumerator<T>` does not support removal of elements from the collection. In C# it is therefore not allowed to remove elements during traversal of a collection in a **foreach** loop. In the Java counterpart to `IEnumerator<T>` (called `Iterator` in Java), there is a `remove` method. The `remove` method can be called once for each step forward in the collection. `remove` is an optional operation in the Java `Iterator` interface. Consequently, removal of elements is not necessarily supported by all implementations of the Java `Iterator` interface.

45.3. The Interface ICollection<T>

Lecture 12 - slide 6

At the next level of Figure 45.1 we encounter the `ICollection<T>` interface. It can be summarized as follows.

- Operations in the interface **`ICollection<T>`**:
  - *The operation prescribed in the superinterface* **`IEnumerable<T>`**
  - bool **`Contains`**(**`T`** element)
  - void **`Add`**(**`T`** element)
  - bool **`Remove`**(**`T`** element)
  - void **`Clear`**()
  - void **`CopyTo`**(**`T[]`** targetArray, int startIndex)
  - int **`Count`**
  - bool **`IsReadOnly`**

In addition to traversability, elements of type `T` can be added to and removed from objects of type `ICollection<T>`. At this level of abstraction, it is not specified where in the collection an element is added. As listed about, a few other operations are supported: Membership testing (`Contains`), resetting (`Clear`), copying of the collection to an array (`CopyTo`), and measuring of size (`Count`). Some collections cannot be mutated once they have been created. The `IsReadOnly` property allows us to find out if a given `ICollection` object is a read only collection.

45.4. The Interface IList<T>

Lecture 12 - slide 7

At the next level of interfaces in Figure 45.1 we meet `IList<T>`. This interface prescribes random access to elements.

- Operations in the interface **`IList<T>`**:
  - *Those prescribed in the superinterfaces* **`ICollection<T>`** *and* **`IEnumerable<T>`**
  - **`T this`**[int index]
  - int **`IndexOf`**(**`T`** element)
  - void **`Insert`**(int index, **`T`** element)
  - void **`RemoveAt`**(int index)

In addition to `ICollection<T>`, the type `IList<T>` allows for indexed access to the `T` elements. The first mentioned operation (`this`) is an indexer, and `IndexOf` is its inverse operation. (See Chapter 19 for a general discussion of indexers). In addition, `IList<T>` has operations for inserting and removing elements at given index positions.

45.5. Overview of the class Collection<T>

Lecture 12 - slide 8

We now encounter the first class in the collection hierarchy, namely `Collection<T>`. Most interfaces and classes discussed in this chapter belong to the namespace `System.Collections.Generic`, but of some odd reason the class `Collection<T>` belongs to `System.Collections.ObjectModel`.

As can be seen from Figure 45.1 the generic class `Collection<T>` implements the generic interface `IList<T>`. As such it supports all the operations of the three interfaces we discussed in Section 45.2 - Section 45.4. As it appears from Figure 45.1 the generic class `List<T>` implements the same interface. It turns out that `Collection<T>` is a minimal class which implements the three interfaces, and not much more. As we will see in Section 45.9, `List<T>` has many more operations, most of which are not prescribed by the interfaces it implement.

Basically, an instance of `Collection<T>` supports indexed access to its elements. Contrary to arrays, however, there is no limit on the number of elements in the collection. The generic class `Collection<T>` has another twist: It is well suited as a superclass for specialized (non-generic) collections. We will see why and how in Section 45.7.

We will not summarize the public interface of `Collection<T>` in the paper version of material, because it is the sum of the interfaces of `IEnumerable<T>`, `ICollection<T>`, and `IList<T>`. You should, however notice the two constructors of `Collection<T>`, a parameterless constructor and a non-copying, "wrapping" constructor on an `IList<T>`.

Collection initializers are new in C# 3.0. Instead of initializing a collection via an `IList`, typically an array, such as in

```
  Collection<int> lst = new Collection<int>(new int[]{1, 2, 3, 4});
```

it is possible in C# 3.0 to make use of collection initializers:

```
  Collection<int> lst = new Collection{1, 2, 3, 4};
```

A collection initializer uses the `Add` method repeatedly to insert the elements within **`{...}`** into an empty list.

Collection initializers are often used in concert with *object initializers*, see Section 18.4, to provide for smooth creation of collection of objects, which are instances of our own types.

You may be interested to know details of the actual representation (data structure) used internally in the generic class `Collection<T>`. Is it an array? Is it a linked list? Or is it something else, such as a mix of arrays and lists, or a tree structure? Most likely, it is a resizeable array. Notice however that from an object-oriented programming point of view (implying encapsulation and visibility control) it is inappropriate to ask such a question. It is sufficient to know about the interface of `Collection<T>` together with the time complexities of the involved operations. (As an additional remark, the source code of the C# libraries written by Microsoft is not generally available for inspection. Therefore we cannot easily check the representation details of the class). The interface of `Collection<T>` includes details about the execution times of the operations of `Collection<T>` relative to the size of a collection. We deal with timing issues of the operations in the collection classes in Section 45.17.

45.6. Sample use of class Collection<T>

Lecture 12 - slide 9

Let us now write a program that shows how to use the central operations in `Collection<T>`. In Program 45.1 we use an instance of the constructed class `Collection<char>`. Thus, we deal with a collection of character values. It is actually worth noticing that we in C# can deal with collections of value types (such as `Collection<char>`) as well as collections of reference types (such as `Collection<Point>`).

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

class BasicCollectionDemo{

  public static void Main(){

    // Initialization - use of a collection initializer. After that add 2 elements.
    IList<char> lst = new Collection<char>{'a', 'b', 'c'};
    lst.Add('d'); lst.Add('e');
    ReportList("Initial List", lst);

    // Mutate existing elements in the list:
    lst[0] = 'z'; lst[1]++;
    ReportList("lst[0] = 'z'; lst[1]++;", lst);

    // Insert and push towards the end:
```

```
19     lst.Insert(0,'n');
20     ReportList("lst.Insert(0,'n');", lst);
21 
22     // Insert at end - with Insert:
23     lst.Insert(lst.Count,'x');      // equivalent to lst.Add('x');
24     ReportList("lst.Insert(lst.Count,'x');", lst);
25 
26     // Remove element 0 and pull toward the beginning:
27     lst.RemoveAt(0);
28     ReportList("lst.RemoveAt(0);", lst);
29 
30     // Remove first occurrence of 'c':
31     lst.Remove('c');
32     ReportList("lst.Remove('c');", lst);
33 
34     // Remove remaining elements:
35     lst.Clear();
36     ReportList("lst.Clear(); ", lst);
37 
38   }
39 
40   public static void ReportList<T>(string explanation, IList<T> list){
41     Console.WriteLine(explanation);
42     foreach(T el in list)
43       Console.Write("{0, 3}", el);
44     Console.WriteLine(); Console.WriteLine();
45   }
46 
47 }
```

Program 45.1 *Basic operations on a Collection of characters.*

The program shown above explains itself in the comments, and the program output in Listing 45.2 is also relatively self-contained. Notice the use of the *collection initializer* in line 9 of Program 45.1. As mentioned in Section 45.5 collection initializers have been introduced in C# 3.0. In earlier versions of C# it was necessary to initialize a collection by use or an *array initializer* (see the discussion of Program 6.7) via the second constructor mentioned above.

```
Initial List
  a  b  c  d  e

lst[0] = 'z'; lst[1]++;
  z  c  c  d  e

lst.Insert(0,'n');
  n  z  c  c  d  e

lst.Insert(lst.Count,'x');
  n  z  c  c  d  e  x

lst.RemoveAt(0);
  z  c  c  d  e  x

lst.Remove('c');
  z  c  d  e  x

lst.Clear();
```

Listing 45.2 *Output of the program with basic operations on a Collection of characters.*

We make the following important observations about the operations in `Collection<T>`:

- The indexer `lst[idx] = expr` mutates an existing element in the collection
    - *The length of the collection is unchanged*
- The `Insert` operation splices a new element into the collection
    - Push subsequent elements towards the end of the collection
    - *Makes the collection longer*
- The `Remove` and `RemoveAt` operations take elements out of the collections
    - Pull subsequent elements towards the beginning of the collection
    - *Makes the collection shorter*

45.7. Specialization of Collections

Lecture 12 - slide 10

Let us now assume that we wish to make our own, specialized (non-generic) collection class of a particular type of objects. Below we will - for illustrative purposes - write a class called `AnimalFarm` which is intended to hold instances of class `Animal`. It is reasonable to program `AnimalFarm` as a subclass of an existing collection class. In this section we shall see that `Collection<Animal>` is a good choice of superclass of `AnimalFarm`.

The class `AnimalFarm` depends on the class `Animal`. You are invited to take a look at class `Animal` via the accompanying slide . We do not include class `Animal` here because it does not add new insight to our interests in collection classes. The four operations of class `AnimalFarm` are shown below.

```
using System;
using System.Collections.ObjectModel;

public class AnimalFarm: Collection<Animal>{

  protected override void InsertItem(int i, Animal a){
    base.InsertItem(i,a);
    Console.WriteLine("**InsertItem: {0}, {1}", i, a);
  }

  protected override void SetItem(int i, Animal a){
    base.SetItem(i,a);
    Console.WriteLine("**SetItem: {0}, {1}", i, a);
  }

  protected override void RemoveItem(int i){
    base.RemoveItem(i);
    Console.WriteLine("**RemoveItem: {0}", i);
  }

  protected override void ClearItems(){
    base.ClearItems();
    Console.WriteLine("**ClearItems");
  }

}
```

Program 45.3 *A class AnimalFarm - a subclass of* **`Collection<Animal>`** *- testing protected members.*

It is important to notice that the four highlighted operations in Program 45.3 are redefinitions of virtual, protected methods in `Collection<Animal>`. Each of the methods activate the similar method in the superclass (this is method combination). In addition, they reveal on standard output that the protected method has been called. A more realistic example of class `AnimalFarm` will be presented in Program 45.6.

The four operations are not part of the client interface of class `AnimalFarm`. They are protected operations. The client interface of `AnimalFarm` is identical to the public operations inherited from `Collection<Animal>`. It means that we use the operations `Add`, `Insert`, `Remove` etc. on instances of class `AnimalFarm`.

We should now understand the role of the four protected operations `InsertItem`, `RemoveItem`, `SetItem`, and `ClearItems` relative to the operations in the public client interface. Whenever an element is inserted into a collection, the protected method `InsertItem` is called. Both `Add` and `Insert` are programmed by use of `InsertItem`. Similarly, both `Remove` and `RemoveAt` are programmed by use of `RemoveItem`. And so on. We see that the major functionality behind the operations in `Collection<T>` is controlled by the four protected methods `InsertItem`, `RemoveItem`, `SetItem`, and `ClearItems`.

```
using System;
using System.Collections.ObjectModel;

class App{

  public static void Main(){

    AnimalFarm af = new AnimalFarm();

    // Populating the farm with Add
    af.Add(new Animal("elephant"));
    af.Add(new Animal("giraffe"));
    af.Add(new Animal("tiger"));
    ReportList("Adding elephant, giraffe, and tiger with Add(...)", af);

    // Additional population with Insert
    af.Insert(0, new Animal("dog"));
    af.Insert(0, new Animal("cat"));
    ReportList("Inserting dog and cat at index 0 with Insert(0, ...)", af);

    // Mutate the animal farm:
    af[1] = new Animal("herring", AnimalGroup.Fish, Sex.Male);
    ReportList("After af[1] = herring", af);

    // Remove tiger
    af.Remove(new Animal("tiger"));
    ReportList("Removing tiger with Remove(...)", af);

    // Remove animal at index 2
    af.RemoveAt(2);
    ReportList("Removing animal at index 2, with RemoveAt(2)", af);

    // Clear the farm
    af.Clear();
    ReportList("Clear the farm with Clear()", af);
  }

  public static void ReportList<T>(string explanation, Collection<T> list){
    Console.WriteLine(explanation);
    foreach(T el in list)
      Console.WriteLine("{0, 3}", el);
    Console.WriteLine(); Console.WriteLine();
  }
}
```

Program 45.4 *A sample client of AnimalFarm - revealing use of protected* **`Collection<Animal>`** *methods.*

Take a close look at the output of Program 45.4 in Listing 45.5. The output explains the program behavior.

```
**InsertItem: 0, Animal: elephant
**InsertItem: 1, Animal: giraffe
**InsertItem: 2, Animal: tiger
Adding elephant, giraffe, and tiger with Add(...)
Animal: elephant
Animal: giraffe
Animal: tiger


**InsertItem: 0, Animal: dog
**InsertItem: 0, Animal: cat
Inserting dog and cat at index 0 with Insert(0, ...)
Animal: cat
Animal: dog
Animal: elephant
Animal: giraffe
Animal: tiger


**SetItem: 1, Animal: herring
After af[1] = herring
Animal: cat
Animal: herring
Animal: elephant
Animal: giraffe
Animal: tiger


**RemoveItem: 4
Removing tiger with Remove(...)
Animal: cat
Animal: herring
Animal: elephant
Animal: giraffe


**RemoveItem: 2
Removing animal at index 2, with RemoveAt(2)
Animal: cat
Animal: herring
Animal: giraffe


**ClearItems
Clear the farm with Clear()
```

Listing 45.5 *Output from sample client of AnimalFarm.*

45.8. Specialization of Collections - a realistic example

Lecture 12 - slide 11

The protected methods in class `AnimalFarm`, as shown in Section 45.7, did only reveal if/when the protected methods were called by other methods. In this section we will show a more realistic example that redefines the four protected methods of `Collection<T>` in a more useful way.

In the example we program the following semantics of the insertion and removal operations of class `AnimalFarm`:

- If we add an animal, an additional animal of the opposite sex is also added.
- Any animal removal or clearing of an animal farm is rejected.

In addition, we add a `GetGroup` operation to `AnimalFarm`, which returns a collection (an sub animal farm) of all animals that belongs to a given group (such as all birds).

The class `Animal` has not been changed, and it still available via accompanying slide.

```
using System;
using System.Collections.ObjectModel;

public class AnimalFarm: Collection<Animal>{

  // Auto insert animal of opposite sex
  protected override void InsertItem(int i, Animal a){
    if(a.Sex == Sex.Male){
      base.InsertItem(i,a);
      base.InsertItem(i, new Animal(a.Name, a.Group, Sex.Female));
    } else {
      base.InsertItem(i,a);
      base.InsertItem(i,new Animal(a.Name, a.Group, Sex.Male));
    }
  }

  // Prevent removal
  protected override void RemoveItem(int i){
    Console.WriteLine("[Removal denied]");
  }

  // Prevent clearing
  protected override void ClearItems(){
    Console.WriteLine("[Clearing denied]");
  }

  // Return all male animals in a given group
  public AnimalFarm GetGroup(AnimalGroup g){
    AnimalFarm res = new AnimalFarm();
    foreach(Animal a in this)
      if (a.Group == g && a.Sex == Sex.Male) res.Add(a);
    return res;
  }

}
```

Program 45.6 *The class AnimalFarm - a subclass of* ***`Collection<Animal>`***.

Notice the way we implement the rejection in `RemoveItem` and `ClearItems`: We do not call the superclass operation.

In Program 45.7 (only on web) we show an `AnimalFarm` client program similar (but not not identical) to Program 45.4. The program output in Listing 45.8 (only on web) reveals the special semantics of the virtual, protected operations from `Collection<T>` - as redefined in Program 45.6.

45.9. Overview of the class List<T>

Lecture 12 - slide 12

We are now going to study the generic class `List<T>`. As it appears from Figure 45.1 both `List<T>` and `Collection<T>` implement the same interface, namely `IList<T>`, see Section 45.4. But as already noticed, `List<T>` offers many more operations than `Collection<T>`.

In the same style as in earlier sections, we provide an overview of the important operations of `List<T>`.

- Constructors
  - `List(), List(IEnumerable<T>), List(int)`
  - Via a *collection initializer*: `new List<T> {t1, t2, ..., tn}`
- Element access
  - this`[int], GetRange(int, int)`
- Measurement
  - `Count, Capacity`
- Element addition
  - `Add(T), AddRange(IEnumerable<T>), Insert(int, T), InsertRange(int, IEnumerable<T>)`
- Element removal
  - `Remove(T), RemoveAll(Predicate<T>), RemoveAt(int), RemoveRange(int, int), Clear()`
- Reorganization
  - `Reverse(), Reverse(int, int),`
    `Sort(), Sort(Comparison<T>),`
    `Sort(IComparer<T>), Sort(int, int, IComparer<T>)`
- Searching
  - `BinarySearch(T), BinarySearch(int, int, T, IComparer<T>), BinarySearch(T, IComparer<T>)`
  - `Find(Predicate<T>), FindAll(Predicate<T>), FindIndex(Predicate<T>),`
    `FindLast(Predicate<T>), FindLastIndex(Predicate<T>), IndexOf(T), LastIndexOf(T)`
- Boolean queries
  - `Contains(T), Exists(Predicate<T>), TrueForAll(Predicate<T>)`
- Conversions
  - `ConvertAll<TOutput>(Converter<T,TOutput>), CopyTo(T[]),`

Compared with `Collection<T>` the class `List<T>` offers sorting, searching, reversing, and conversion operations. `List<T>` also has a number of "range operations" which operate on a number of elements via a single operation. We also notice a number of *higher-order operations*: Operations that take a delegate value (a function) as parameter. `ConvertAll` is a generic method which is parameterized with the type `TOutput`. `ConvertAll` accepts a function of delegate type which converts from type `T` to `TOutput`.

45.10. Sample use of class List<T>

Lecture 12 - slide 13

In this and the following sections we will show how to use some of the operations in `List<T>`. We start with a basic example similar to Program 45.1 in which we work on a list of characters: `List<char>`. We insert a number of `char` values into a list, and we remove some values as well. The program appears in Program 45.9 and the self-explaining output can be seen in Listing 45.10 (only on web). Notice in particular how the range operations `InsertRange` (line 28) and `RemoveRange` (line 40) operate on the list.

```
1  using System;
2  using System.Collections.Generic;
3  
4  /* Very similar to our illustration of class Collection<char> */
5  class BasicListDemo{
6  
7    public static void Main(){
8  
9      // List initialization and adding elements to the end of the list:
10     List<char> lst = new List<char>{'a', 'b', 'c'};
11     lst.Add('d'); lst.Add('e');
12     ReportList("Initial List", lst);
13 
14     // Mutate existing elements in the list
15     lst[0] = 'z'; lst[1]++;
16     ReportList("lst[0] = 'z'; lst[1]++;", lst);
17 
18     // Insert and push towards the end
19     lst.Insert(0,'n');
20     ReportList("lst.Insert(0,'n');", lst);
21 
22     // Insert at end - with Insert
23     lst.Insert(lst.Count,'x');      // equivalent to lst.Add('x');
24     ReportList("lst.Insert(lst.Count,'x');", lst);
25 
26     // Insert a new list into existing list, at position 2.
27     lst.InsertRange(2, new List<char>{'1', '2', '3', '4'});
28     ReportList("lst.InsertRange(2, new List<char>{'1', '2', '3', '4'});", lst);
29 
30     // Remove element 0 and push toward the beginning
31     lst.RemoveAt(0);
32     ReportList("lst.RemoveAt(0);", lst);
33 
34     // Remove first occurrence of 'c'
35     lst.Remove('c');
36     ReportList("lst.Remove('c');", lst);
37 
38     // Remove 2 elements, starting at element 1
39     lst.RemoveRange(1, 2);
40     ReportList("lst.RemoveRange(1, 2);", lst);
41 
42     // Remove all remaining digits
43     lst.RemoveAll(delegate(char ch){return Char.IsDigit(ch);});
44     ReportList("lst.RemoveAll(delegate(char ch){return Char.IsDigit(ch);});", lst);
45 
46     // Test of all remaining characters are letters
47     if (lst.TrueForAll(delegate(char ch){return Char.IsLetter(ch);}))
48       Console.WriteLine("All characters in lst are letters");
49     else
50       Console.WriteLine("NOT All characters in lst are letters");
51   }
```

```

  public static void ReportList<T>(string explanation, List<T> list){
    Console.WriteLine(explanation);
    foreach(T el in list)
      Console.Write("{0, 3}", el);
    Console.WriteLine(); Console.WriteLine();
  }

}
```

Program 45.9 *Basic operations on a List of characters.*

45.11. Sample use of the Find operations in List<T>

Lecture 12 - slide 14

In this section we will illustrate how to use the search operations in `List<T>`. More specifically, we will apply the methods `Find`, `FindAll` and `IndexOf` on an instance of `List<Point>`, where `Point` is a type, such as defined by the struct in Program 14.12. The operations discussed in this section do all use linear search. It means that they work by looking at one element after the other, in a rather trivial way. As a contrast, we will look at binary search operations in Section 45.13, which searches in a "more advanced" way.

In the program below - Program 45.11 - we declare a `List<Point>` in line 11, and we add six points to the list in line 13-16. In line 20 we shown how to use `Find` to locate the first point in the list whose x-coordinate is equal to 5. The same is shown in line 25. The difference between the two uses of `Find` is that the first relies on a delegate given on the fly: `delegate(Point q){return (q.Getx() == 5);}`, while the other relies on an existing static method `FindX5` (defined in line 40 - 42). The approach shown in line 20 is, in my opinion, superior.

In line 29 we show how to use the variant `FindAll`, which returns a `Point` list instead of just a single `Point`, as returned by `Find`. In line 36 we show how `IndexOf` can be used to find the index of a given `Point` in a `Point` list. It is worth asking how the `Point` parameter of `IndexOf` is compared with the points in `Point` list. The documentation states that the points are compared by use of the default equality comparer of the type `T`, which in our case is struct `Point`. We have discussed *the default equality comparer* in Section 42.9 in the slipstream of our coverage of the generic interfaces `IEquatable<T>` and `IEqualityComparer<T>`.

We use the static method `ReportList` to show a `Point` list on standard output. We call `ReportList` several times in Program 45.11. The program output is shown in Listing 45.12.

```
1  using System;
2  using System.Collections.Generic;
3  
4  class C{
5  
6    public static void Main(){
7  
8       System.Threading.Thread.CurrentThread.CurrentCulture =
9          new System.Globalization.CultureInfo("en-US");
10 
11      List<Point> pointLst = new List<Point>();
12 
13      // Construct points and point list:
14      pointLst.Add(new Point(0,0)); pointLst.Add(new Point(5, 9));
15      pointLst.Add(new Point(5,4)); pointLst.Add(new Point(7.1,-13));
16      pointLst.Add(new Point(5,-2)); pointLst.Add(new Point(14,-3.4));
```

```
    ReportList("Initial point list", pointLst);

    // Find first point in list with x coordinate 5
    Point p = pointLst.Find(delegate(Point q){return (q.Getx() == 5);});
    Console.WriteLine("Found with delegate predicate: {0}\n", p);

    // Equivalent. Use predicate which is a static method
    p = pointLst.Find(new Predicate<Point>(FindX5));
    Console.WriteLine("Found with static member predicate: {0}\n", p);

    // Find all points in list with x coordinate 5
    List<Point> resLst = new List<Point>();
    resLst = pointLst.FindAll(delegate(Point q){return (q.Getx() == 5);});
    ReportList("All points with x coordinate 5", resLst);

    // Find index of a given point in pointLst.
    // Notice that Point happens to be a struct - thus value comparison
    Point searchPoint = new Point(5,4);
    Console.WriteLine("Index of {0} {1}", searchPoint,
                       pointLst.IndexOf(searchPoint));

  }

  public static bool FindX5(Point p){
    return p.Getx() == 5;
  }

  public static void ReportList<T>(string explanation,List<T> list){
    Console.WriteLine(explanation);
    int cnt = 0;
    foreach(T el in list){
      Console.Write("{0, 3}", el);
      cnt++;
      if (cnt%4 == 0) Console.WriteLine();
    }
    if (cnt%4 != 0) Console.WriteLine();
    Console.WriteLine();
  }
}
```

Program 45.11 *Sample uses of List.Find.*

```
Initial point list
Point:(0,0). Point:(5,9). Point:(5,4). Point:(7.1,-13).
Point:(5,-2). Point:(14,-3.4).

Found with delegate predicate: Point:(5,9).

Found with static member predicate: Point:(5,9).

All points with x coordinate 5
Point:(5,9). Point:(5,4). Point:(5,-2).

Index of Point:(5,4).  2
```

Listing 45.12 *Output from the Find program.*

45.12. Sample use of Sort in List<T>

Lecture 12 - slide 15

As a client user of the generic class `List<T>` it is likely that you never need to write a sorting procedure! You are supposed to use one of the already existing `Sort` methods in `List<T>`.

Sorting the elements in a collection of elements of type `T` depends on a *less than or equal operation* on `T`. If the type `T` is taken directly from the C# libraries, it may very well be the case that we can just use the default *less than or equal operation* of the type `T`. If `T` is one of our own types, we will have to supply an implementation of the comparison operation ourselves. This can be done by passing a delegate object to the `Sort` method.

Below, in Program 45.13 we illustrate most of the four overloaded `Sort` operations in `List<T>`. The actual type parameter in the example, passed for `T`, is `int`. The program output (the lists before and after sorting) is shown in Listing 45.14 (only on web).

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

class C{

  public static void Main(){

     List<int> listOriginal = new List<int>{5, 3, 2, 7, -4, 0},
               list;

     // Sorting by means of the default comparer of int:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort();
     ReportList(list);
     Console.WriteLine();

     // Equivalent - explicit notatation of the Comparer:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort(Comparer<int>.Default);
     ReportList(list);
     Console.WriteLine();

     // Equivalent - explicit instantiation of an IntComparer:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort(new IntComparer());
     ReportList(list);
     Console.WriteLine();

     // Similar - use of a delegate value for comparison:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort(delegate(int x, int y){
                 if (x < y)
                    return -1;
                 else if (x == y)
                    return 0;
                 else return 1;});
     ReportList(list);
     Console.WriteLine();
  }

45   public static void ReportList<T>(List<T> list){
46     foreach(T el in list)
47       Console.Write("{0, 3}", el);
48     Console.WriteLine();
49   }
50
51 }
52
53 public class IntComparer: Comparer<int>{
54   public override int Compare(int x, int y){
55     if (x < y)
56       return -1;
57     else if (x == y)
58       return 0;
59     else return 1;
60   }
61 }
```

Program 45.13 *Four different activations of the List.Sort method.*

Throughout Program 45.13 we do several sortings of `listOriginal`, as declared in line 8. In line 14 we rely the default comparer of type `int`. The default comparer is explained in the following way in the .NET framework documentation of `List.Sort`:

> This method uses the default comparer `Comparer.Default` for type `T` to determine the order of list elements. The `Comparer.Default` property checks whether type `T` implements the `IComparable` generic interface and uses that implementation, if available. If not, `Comparer.Default` checks whether type `T` implements the `IComparable` interface. If type `T` does not implement either interface, `Comparer.Default` throws an `InvalidOperationException`.

The sorting done in line 21 is equivalent to line 14. In line 21 we show how to pass *the default comparer* of type `int` explicitly to the `Sort` method.

Let us now assume the type `int` does not have a default comparer. In other words, we will have to implement the comparer ourselves. The call of `Sort` in line 28 passes a new `IntComparer` instance to `Sort`. The class `IntComparer` is programmed in line 53-61, at the bottom of Program 45.13. Notice that `IntComparer` is a subclass of `Comparer<int>`, which is an abstract class in the namespace `System.Collections.Generic`with an abstract method named `Compare`. The generic class `Comparer<T>` is in many ways similar to the class `EqualityComparer<T>`, which we touched on in Section 42.9. Most important, both have a static `Default` property, which returns a comparer object.

As a final resort that always works we can pass a comparer function to `Sort`. In C#, such a function is programmed as a delegate. (Delegates are discussed in Chapter 22). Line 35-40 shows how this can be done. Notice that the delegate we use is programmed on the fly. This style of programming is a reminiscence of *functional programming*.

I find it much more natural to pass an *ordering method* instead of *an object of a class with an ordering method*. (The latter is a left over from older object-oriented programming languages in which the only way to pass a function `F` as parameter is via an object of a class in which `F` is an instance method). In general, I also prefer to be explicit about the ordering instead of relying on some default ordering which may turn out to surprise you.

Let us summarize the lessons that we have learned from the example:

- Some types have a default comparer which is used by **`List.Sort()`**
- The default comparer of T can extracted by **`Comparer<T>.Default`**
- An *anonymous delegate comparer* is attractive if the default comparer of the type does not exist, of if it is inappropriate.

Exercise 12.1. *Shuffle List*

Write a `Shuffle` operation that disorders the elements of a collection in a random fashion. A shuffle operation is useful in many context. There is no `Shuffle` operation in **`System.Collections.Generic.List<T>`**. In the similar Java libraries there is a shuffle method.

In which class do you want to place the `Shuffle` operation? You may consider to make use of extension methods.

You can decide on programming either a mutating or a non-mutating variant of the operation. Be sure to understand the difference between these two options.

Test the Shuffle operation, for instance on **`List<Card>`**. The class `Card` (representing a playing card) is one of the classes we have seen earlier in the course.

Exercise 12.2. *Course and Project classes*

In the earlier exercise about courses and projects (found in the lecture about abstract classes and interfaces) we refined the program about `BooleanCourse`, `GradedCourse`, and `Project`. Revise your solution (or the model solution) such that the courses in the class `Project` are represented as a variable of type `List<Course>` instead of by use of four variables of type `Course`.

Reimplement and simplify the method `Passed` in class `Project`. Take advantage of the new representation of the courses in a project, such that the "3 out of 4 rule" (see the original exercise) is implemented in a more natural way.

45.13. Sample use of BinarySearch in List<T>

Lecture 12 - slide 16

The search operations discussed in Section 45.11 all implemented *linear search* processes. The search operations of this section implement *binary search* processes, which are much faster when applied on large collections. On collections of size *n*, linear search has - not surprisingly - time complexity *O(n)*. Binary search has time complexity *O(log n)*. When *n* is large, the difference between *n* and *log n* is dramatic.

The `BinarySearch` operations in `List<T>` require, as a precondition, that the list is ordered before the search is performed. If necessary, the `Sort` operation (see Section 45.12) can be used to establish the ordering.

You may ask why we should search for an element which we - in the starting point - is able to pass as input to the `BinarySearch` method. There is a couple of good answers. First, we may be interested to know if the element is present or not in the list. Second, it may also be possible to search for an incomplete object (by only comparing some selected fields in the `Comparer` method). Using this approach we are actually interested in finding the complete object, with all the data fields, in the collection.

If the `BinarySearch` operation finds an element in the list, the index of the element is returned. This is a non-negative integer. If the element is not found, a negative integer, say *i*, is returned. Below we will see that that -*i* (or more precisely the bitwise complement ~*i*) in that case is the position of the element, if it had been present in the list.

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

class BinarySearchDemo{

  public static void Main(){

     System.Threading.Thread.CurrentThread.CurrentCulture =
        new System.Globalization.CultureInfo("en-US");

     List<Point> pointLst = new List<Point>();  // Point is a struct.

     // Construct points and point list:
     pointLst.Add(new Point(0,0)); pointLst.Add(new Point(5, 9));
     pointLst.Add(new Point(5,4)); pointLst.Add(new Point(7.1,-13));
     pointLst.Add(new Point(5,-2)); pointLst.Add(new Point(14,-3.4));
     ReportList("The initial point list", pointLst);

     // Sort point list, using a specific point Comparer.
     // Notice the PointComparer:
     // Ordering according to sum of x and y coordinates
     IComparer<Point> pointComparer = new PointComparer();
     pointLst.Sort(pointComparer);
     ReportList("The sorted point list", pointLst);

     int res;
     Point searchPoint;

     // Run-time error.
     // Failed to compare two elements in the array.
//   searchPoint = new Point(5,4);
//   res = pointLst.BinarySearch(searchPoint);
//   Console.WriteLine("BinarySearch for {0}: {1}", searchPoint, res);

     searchPoint = new Point(5,4);
     res = pointLst.BinarySearch(searchPoint, pointComparer);
     Console.WriteLine("BinarySearch for {0}: {1}", searchPoint, res);

     searchPoint = new Point(1,8);
     res = pointLst.BinarySearch(searchPoint, pointComparer);
     Console.WriteLine("BinarySearch for {0}: {1}", searchPoint, res);

  }

  public static void ReportList<T>(string explanation,List<T> list){
    Console.WriteLine(explanation);
    int cnt = 0;
    foreach(T el in list){
      Console.Write("{0, 3}", el);
      cnt++;
      if (cnt%4 == 0) Console.WriteLine();
    }
    if (cnt%4 != 0) Console.WriteLine();
    Console.WriteLine();
  }

}

// Compare the sum of the x and y coordinates.
```

```
60 // Somewhat non-traditional!
61 public class PointComparer: Comparer<Point>{
62   public override int Compare(Point p1, Point p2){
63     double p1Sum = p1.Getx() + p1.Gety();
64     double p2Sum = p2.Getx() + p2.Gety();
65     if (p1Sum < p2Sum)
66       return -1;
67     else if (p1Sum == p2Sum)
68       return 0;
69     else return 1;
70   }
71 }
```

Program 45.15 *Sample uses of List.BinarySearch.*

Program 45.15 works on a list of points. Six points are created and inserted into a list in line 13-16. Next, in line 23, the list is sorted. As it appears from the `Point` comparer programmed in line 62-72, a point *p* is less than or equal to point *q*, if *p*.x + *p*.y <= *q*.x + *q*.y. You may think that this is odd, but it is our decision for this particular program example.

In line 33 we attempt to activate binary searching by use of the default comparer. But such a comparer does not exist for class *Point*. This problem is revealed at run-time.

In line 37 and 41 we search for the points (5,4) and (1,8) respectively. In both cases we expect to find the point (5,4), which happens to be located at place 3 in the sorted list. The output of the program, shown in Program 45.17 (only on web) confirms this.

In the next program, Program 45.17 we illustrate what happens if we search for a non-existing point with `BinarySearch`. The class `PointComparer` and the generic method `ReportList` are not shown in the paper version of Program 45.17. Please consult Program 45.15 where they both appear.

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

class BinarySearchDemo{

  public static void Main(){

     System.Threading.Thread.CurrentThread.CurrentCulture =
        new System.Globalization.CultureInfo("en-US");

     List<Point> pointLst = new List<Point>();

     // Construct points and point list:
     pointLst.Add(new Point(0,0)); pointLst.Add(new Point(5, 9));
     pointLst.Add(new Point(5,4)); pointLst.Add(new Point(7.1,-13));
     pointLst.Add(new Point(5,-2)); pointLst.Add(new Point(14,-3.4));
     ReportList("Initial point list", pointLst);

     // Sort point list, using a specific point Comparer:
     IComparer<Point> pointComparer = new PointComparer();
     pointLst.Sort(pointComparer);
     ReportList("Sorted point list", pointLst);

     int res;
     Point searchPoint;

     searchPoint = new Point(1,1);
     res = pointLst.BinarySearch(searchPoint, pointComparer);
     Console.WriteLine("BinarySearch for {0}: {1}\n", searchPoint, res);
30
31     if (res < 0){    // search point not found
32       pointLst.Insert(~res, searchPoint);  // Insert searchPoint such
33                                            // that pointLst remains sorted
34       Console.WriteLine("Inserting {0} at index {1}", searchPoint, ~res);
35       ReportList("Point list after insertion", pointLst);
36     }
37   }
38
39   // ReportList not shown
40 }
41
42 // Class PointComparer not shown
```

Program 45.17 *Searching for a non-existing Point.*

The scene of Program 45.17 is the same as that of Program 45.15. In line 28 we do binary searching, looking for the point (1,1). None of the points in the program have an "x plus y sum" of 2. Therefore, the point (1,1) is not located by `BinarySearch`. The `BinarySearch` method returns a negative *ghost index*. The ghost index is the bitwise complement of the index where to insert the point in such a way that the list will remain sorted. (Notice the bitwise complement operation **~** which turns 0 to 1 and 1 to 0 at the binary level). The program output reveals that position ~(-3) is the natural place of the point (1,1) to maintain the ordering of the list. Notice that the value of ~(-3) is 2, due the use of two's complement arithmetic. This explains the rationale of the negative values returned by `BinarySearch`.

The output of Program 45.17 is shown in Listing 45.18 (only on web).

Contrary to `Sort`, it is not possible to pass a delegate to `BinarySearch`. This seems to be a flaw in the design of the `List<T>` library.

We have learned the following lessons about `BinarySearch`:

- Binary search can only be done on sorted lists
- In order to use binary search, we need - in general - to provide an explicit `Comparer` object
- Binary search returns a (non-negative) integer if the element is found
    - The index of the located element
- Binary search returns a negative integer if the element is not found
    - The complement of this number is a *ghost index*
    - The index of the element if it had been in the list

45.14. Overview of the class LinkedList<T>

Lecture 12 - slide 17

The collections implemented by `Collection<T>` of Section 45.5 and `List<T>` of Section 45.9 were based on arrays. We will now turn our interest towards a list type, which is based on a *linked* representation.

Below, in Figure 45.2 we show the object-structure of a double linked list.

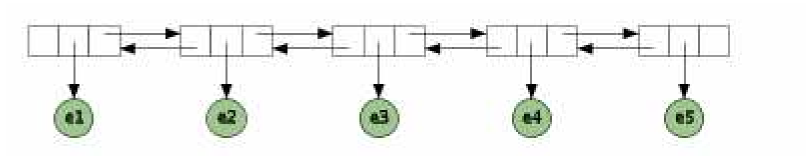

Figure 45.2 *A double linked list where instances of `LinkedListNode` keep the list together*

The generic class `LinkedList<T>` relies on a "building block class" `LinkedListNode<T>`. We need to deal with instances of `LinkedListNodes` when we work with linked lists in C#. In other words, `LinkedListNode` is not just a class behind the scene - it is an important class for clients of `LinkedListNode<T>`. In Figure 45.2 the five rectangular nodes are instances of `LinkedListNode<T>` for some element type `T`. The circular, green nodes are instances of the element type `T`. We will study `LinkedListNode<T>` in Section 45.15 after we have surveyed the list operations in `LinkedList<T>`.

As it can be seen from the class diagram of the list class in Figure 45.1, `LinkedList<T>` implements the interface `ICollection<T>`, see Section 45.3. Unlike `Collection<T>` and `List<T>`, `LinkedList<T>` does not implement indexed access, as of `Ilist<T>`. This is a natural choice because indexed access is not efficient in a linked representation. The following operations are available in `LinkedList<T>`:

- Constructors
  - `LinkedList(), LinkedList(IEnumerable<T>)`
- Accessors (properties)
  - `First`, `Last`, `Count`
- Element addition
  - `AddFirst(T), AddFirst(LinkedListNode<T>), AddLast(T), AddLast(LinkedListNode<T>), AddBefore(LinkedListNode<T>, T), AddBefore(LinkedListNode<T>, LinkedListNode<T>), AddAfter(LinkedListNode<T>, T), AddAfter(LinkedListNode<T>, LinkedListNode<T>), Add(T)`
- Element removal
  - `Remove(T), Remove(LinkedListNode<T>), RemoveFirst(), RemoveLast(), Clear()`
- Searching
  - `Find(T), FindLast(T)`
- Boolean queries
  - `Contains(T)`

A linked list can be constructed as an empty collection or as a collection filled with elements from another collection, represented as an `IEnumerable<T>`, see Section 45.2.

The `First` and `Last` properties access the first/last `LinkedListNode` in the double linked list. `Count` returns the number of elements in the list - not by counting them each time `Count` is referred - but via some bookkeeping information encapsulated in a linked list object. Thus, `Count` is an *O(1)* operation.

Although `LinkedList<T>` implements the generic interface `ICollection<T>`, which has a method named `Add`, the `Add` operation is not readily available on linked lists. We will in Program 45.19 show that `Add` is present as an explicit interface implementation, see Section 31.8. Instead of `Add`, the designers of `LinkedList<T>` want us to use one of the `Add`*Relative* operations: `AddFirst`, `AddLast`, `AddBefore`, and `AddAfter`. None of these are prescribed by the interface `ICollection<T>`, however. Each of the `Add`*Relative* operations are overloaded in two variants, such that we can add an element of type `T` or an object of type `LinkedListNode<T>` (which in turn reference an object of type `T`).

Using the `Remove` methods, it is possible to remove an element of type T or a specific instance of `LinkedListNode<T>`. `Remove(T)` is an *O(n)* operation; `Remove(LinkedListNode<T>)` is an *O(1)* operation. There are also parameter-less methods for removing the first/last element in the linked list. The time complexity of these are *O(1)*.

Finally there are linear search operations from either end of the list: `Find` and `FindLast`. The boolean `Contains` operation is similar to the `Find` operations. These operations all seem to rely on the `Equals` operation inherited from class `Object`. In that way `Find`, `FindLast` and `Contains` are more primitive (not as well-designed) as the similar methods in `List<T>`. (The documentation in the .NET libraries is silent about these details).

45.15. The class LinkedListNode<T>

Lecture 12 - slide 18

As illustrated in Figure 45.2, instances of the generic class `LinkedListNode<T>` keep a linked list together. In the figure, the rectangular boxes are instances of `LinkedListNode<T>`. From the figure it appears that each instance of `LinkedListNode<T>` has three references: One to the left, one to the element, and one to the right. Actually, there is a fourth reference, namely to the linked list instance to which a given `LinkedListNode` object belongs.

The class **`LinkedListNode<T>`** is sealed, generic class that represents a non-mutable node in a linked list

A `LinkedListNode` can at most belong to a single linked list

The members of `LinkedListNode<T>` are as follows:

- A single constructor **`LinkedListNode(T)`**
- Four properties
  - **`Next`** - getter
  - **`Previous`** - getter
  - **`List`** - getter
  - **`Value`** - getter and setter

The properties `Next` and `Previous` access neighbor instances of `LinkedListNode<T>`. `Value` accesses the element of type `T`. `List` accesses the linked list to which the instance of `LinkedListNode` belongs. `Next`, `Previous`, and `List` are all getters. `Value` is both a getter and a setter.

It is not possible to initialize or to mutate the fields behind the properties `Next`, `Previous`, and `List` via public interfaces. It is clearly the intention that the linked list - and only linked list - has authority to change these fields. If we programmed our own, special-purpose linked list class it would therefore not be easy to reuse the class `LinkedListNode<T>`. This is unfortunate.

Related to the discussion about the interface of `LinkedListNode<T>` we may ask how `LinkedList` is allowed to access the private/internal details of an instance of `LinkedListNode`. The best guess seems to be that the fields are internal.

45.16. Sample use of class LinkedList<T>

Lecture 12 - slide 19

We will illustrate the use of `LinkedList<T>` and `LinkedListNode<T>` in Program 45.19. In line 8 we make a linked list of integers from an array. Notice the use of the `LinkedList` constructor `LinkedList(IEnumerable<T>)`.

In line 16 we attempt to add the integer 17 to the linked list. This is not possible, because the method `Add` is not easily available, see the discussion in Section 45.14. If we insist to use `Add`, it must be done as in line 20. Most likely, you should use one of the `Add` variants instead, for instance `AddFirst` or `AddLast`.

```
1  using System;
2  using System.Collections.Generic;
3
4  class LinkedListDemo{
5
6    public static void Main(){
7
8       LinkedList<int> lst = new LinkedList<int>(
9                                  new int[]{5, 3, 2, 7, -4, 0});
10
11      ReportList("Initial LinkedList", lst);
12
13      // Using Add.
14      // Compile-time error: 'LinkedList<int>' does not contain a
15      //                                  definition for 'Add'
16      // lst.Add(17);
17      // ReportList("lst.Add(17);" lst);
18
19      // Add is implemented as an explicit interface implementation
20      ((ICollection<int>)lst).Add(17);
21      ReportList("((ICollection<int>)lst).Add(17);", lst);
22
23      // Using AddFirst and AddLast
24      lst.AddFirst(-88);
25      lst.AddLast(88);
26      ReportList("lst.AddFirst(-88); lst.AddFirst(88);", lst);
27
28      // Using Remove.
29      lst.Remove(17);
30      ReportList("lst.Remove(17);", lst);
31
32      // Using RemoveFirst and RemoveLast
33      lst.RemoveFirst(); lst.RemoveLast();
34      ReportList("lst.RemoveFirst(); lst.RemoveLast();", lst);
35
36      // Using Clear
37      lst.Clear();
38      ReportList("lst.Clear();", lst);
39
40   }
41
42   public static void ReportList<T>(string explanation, LinkedList<T> list){
43     Console.WriteLine(explanation);
```

```
    foreach(T el in list)
      Console.Write("{0, 4}", el);
    Console.WriteLine();  Console.WriteLine();
  }

}
```

Program 45.19   *Basic operations on a LinkedList of integers.*

The output of Program 45.19 is shown in Listing 45.20. By studying Listing 45.20 you will learn additional details of the `LinkedList` operations.

```
Initial LinkedList
   5   3   2   7  -4   0

((ICollection<int>)lst).Add(17);
   5   3   2   7  -4   0  17

lst.AddFirst(-88); lst.AddFirst(88);
 -88   5   3   2   7  -4   0  17  88

lst.Remove(17);
 -88   5   3   2   7  -4   0  88

lst.RemoveFirst(); lst.RemoveLast();
   5   3   2   7  -4   0

lst.Clear();
```

Listing 45.20   *Output of the program with basic operations on a LinkedList.*

The `LinkedList` example in Program 45.19 did not show how to use `LinkedListNodes` together with `LinkedList<T>`. To make up for that we will in Program 45.21 concentrate on the use of `LinkedList<T>` and `LinkedListNode<T>` together.

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

class LinkedListNodeDemo{

  public static void Main(){

     LinkedList<int> lst = new LinkedList<int>(
                            new int[]{5, 3, 2, 7, -4, 0});
     ReportList("Initial LinkedList", lst);

     LinkedListNode<int> node1, node2, node;
     node1 = lst.First;
     node2 = lst.Last;

     // Run-time error.
     // The LinkedListNode is already in the list.
     // Error message: The LinkedList node belongs a LinkedList.
/*   lst.AddLast(node1);    */

     // Move first node to last node in list
     lst.Remove(node1); lst.AddLast(node1);
     ReportList("node1 = lst.First; lst.Remove(node1); lst.AddLast(node1);", lst);

     // Navigate in list via LinkedListNode objects
26      node1 = lst.First;
27      Console.WriteLine("Third element in list: node1 = lst.First;
28 node1.Next.Next.Value    {0}\n",
29                         node1.Next.Next.Value);
30
31      // Add an integer after a LinkedListNode object
32      lst.AddAfter(node1, 17);
33      ReportList("lst.AddAfter(node1, 17);", lst);
34
35      // Add a LinkedListNode object after another LinkedListNode object
36      lst.AddAfter(node1, new LinkedListNode<int>(18));
37      ReportList("lst.AddAfter(node1, new LinkedListNode<int>(18));" , lst);
38
39      // Navigate in LinkedListNode objects and add an int before a node:
40      node = node1.Next.Next.Next;
41      lst.AddBefore(node, 99);
42      ReportList("node = node1.Next.Next.Next; lst.AddBefore(node, 99); " , lst);
43
44      // Navigate in LinkedListNode objects and remove a node.
45      node = node.Previous;
46      lst.Remove(node);
47      ReportList("node = node.Previous; lst.Remove(node);" , lst);
48
49   }
50
51   // Method ReportList not shown in this version.
   }
```

Program 45.21 *Basic operations on a LinkedList of integers - using* ***LinkedListNode****s.*

In line 8-9 we make the same initial integer list as in Program 45.19. In line 13-14 we see how to access to the first/last `LinkedListNode` objects of the list.

In line 19 we attempt to add `node1`, which is the first `LinkedListNode` in `lst`, as the last node of the list. This fails because it could bring the linked list into an inconsistent state. (Recall in this context that a `LinkedListNode` knows the list to which it belongs). Instead, as shown in line 22, we should first remove `node1` and then add `node1` with `AddLast`.

Please take a close look at the remaining addings, navigations, and removals in Program 45.21. As above, we show a self-explaining output of the program, see Listing 45.22.

```
Initial LinkedList
   5   3   2   7  -4   0

node1 = lst.First; lst.Remove(node1); lst.AddLast(node1);
   3   2   7  -4   0   5

Third element in list: node1 = lst.First;  node1.Next.Next.Value    7

lst.AddAfter(node1, 17);
   3  17   2   7  -4   0   5

lst.AddAfter(node1, new LinkedListNode<int>(18));
   3  18  17   2   7  -4   0   5

node = node1.Next.Next.Next; lst.AddBefore(node, 99);
   3  18  17  99   2   7  -4   0   5

node = node.Previous; lst.Remove(node);
   3  18  17   2   7  -4   0   5
```

Listing 45.22 *Output of the program with LinkedListNode operations on a LinkedList.*

45.17. Time complexity overview: Collection classes

Lecture 12 - slide 20

In this section we will discuss the efficiency of selected and important list operations in the three classes `Collection<T>`, `List<T>`, and `LinkedList<T>`. This is done by listing the *time complexities* of the operations in a table, see Table 45.1. If you are not comfortable with Big O notation, you can for instance consult Wikipedia [Big-O] or a book about algorithms and data structures.

The time complexities of the list operations are most often supplied as part of the documentation of the operations. The choice of one list type in favor of another is often based on requirements to the time complexities of important operations. Therefore you should pay careful attention to the information about time complexities in the C# library documentation.

Throughout the discussion we will assume that the lists contain *n* elements. It may be helpful to relate the table with the class diagram in Figure 45.1 from which it appears which interfaces to expect from the list classes.

Operation	**`Collection<T>`**	**`List<T>`**	**`LinkedList<T>`**
`this[i]`	*O(1)*	*O(1)*	-
`Count`	*O(1)*	*O(1)*	*O(1)*
`Add(e)`	*O(1) or O(n)*	*O(1) or O(n)*	*O(1)*
`Insert(i,e)`	*O(n)*	*O(n)*	-
`Remove(e)`	*O(n)*	*O(n)*	*O(n)*
`IndexOf(e)`	*O(n)*	*O(n)*	-
`Contains(e)`	*O(n)*	*O(n)*	*O(n)*
`BinarySearch(e)`	-	*O(log n)*	-
`Sort()`	-	*O(n log n) or O(n^2)*	-
`AddBefore(lln)`	-	-	*O(1)*
`AddAfter(lln,e)`	-	-	*O(1)*
`Remove(lln)`	-	-	*O(1)*
`RemoveFirst()`	-	-	*O(1)*
`RemoveLast()`	-	-	*O(1)*

Table 45.1 *Time complexities of important operations in the classes* `Collection<T>`, `List<T>`, *and* `LinkedList<T>`.

As it can be seen in the class diagram of Figure 45.1 all three classes implement the `ICollection<T>` interface with the operations `Count`, `Add`, `Remove`, and `Contains`. Thus, these four operations appear for all classes in Table 45.1.

`Count` is efficient for all lists, because it maintains an internal counter, the value of which can be returned by the `Count` property. Thus, independent of the length of a list, `Count` runs in constant time.

For all three types of lists, `Add(e)` adds an element `e` (of type `T`) to the end of the list. This can be done in constant time, because all the three types of lists have direct access the rear end of the list. The time complexity *O(1)*/*O(n)* given for `Collection<T>` and `List<T>` reflects that under normal circumstances it takes only constant time to add an element to a `Collection` or a `List`. If however, the list is full it may need resizing, and in that case the run time is linear in *n*.

`Remove(e)` and `Contains(e)`, where `e` is of type `T`, will have to search for `e` in the list. This behavior is common for all three types of lists. Therefore the run times of `Remove` and `Contains` are *O(n)*.

The indexer `this[i]` is only available in the lists that implement `Ilist<T>`. Such lists are based on arrays, and therefore the runtime of the indexer is *O(1)*. (Recall that in arrays it is possible to compute the location of an element with a given index; No searching, whatsoever, is involved).

`BinarySearch` and `Sort` are operations in `List<T>`. `Sort` implements a Quicksort variant, and as such the worst possible time complexity is *O(n^2)*, but the expected time complexity is *O(n log n)*. The runtime of `BinarySearch` is, as expected, *O(log n)*.

The bottom five operations in the table belong to `LinkedList`. The methods `AddBefore`, `AddAfter`, and `Remove` all work on a `LinkedListNode`, `lln`, and as such their runtimes do not depend on *n*. (Only a few references need to be assigned. The number of pointer assignments do not depend on *n*). Thus, when applied on objects of type `LinkedListNode` the runtime of these three operations are *O(1)*. `RemoveFirst` and `RemoveLast` are of time complexity *O(1)* because a linked list maintain direct references to both ends of the list.

45.18. Using Collections through Interfaces

Lecture 12 - slide 21

We started this chapter with a discussion of list interfaces, and we will end the chapter in a similar way.

It is, of course, necessary to use one of the collection classes (such as `List<T>`) when you need a collection in your program. The morale of this section is, however, that you should not use list classes more than necessary. In short, you should typically use `List<T>` or `Collection<T>` (for some type T) when you make a collection object. All other places you are better off using one of the interface types, such as `IList<T>`. The key observations can be summarized as follows.

It is an advantage to use collections via interfaces instead of classes

If possible, only use collection classes in instantiations, just after `new`

This leads to programs with fewer bindings to concrete implementations of collections

With this approach, it is easy to replace a collection class with another

Thus, please consider the following when you use collections:

Program against collection interfaces, not collection classes

If the types of variables and parameters are given as interfaces it is easy, a later point in time, to change the representation of your collections (say, from `Collection<T>` to one of your own collections which implements `Ilist<T>`). Notice that if you, for instance, apply `List<T>` operations, which are not prescribed by one of the interfaces, you need to declare your list of type `List<T>` for some type `T`.

Let us illustrate how this can be done in Program 45.23. The thing to notice is that the only place we refer to a list class (here `Collection<Animal>`()) is in line 9: new `Collection<Animal>`. All other places, as emphasized with **purple**, we use the interface `ICollection<Animal>`. If we, tomorrow, wish to change the representation of the animal collection, the only place to modify is line 9.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4  
5  
6  class CollectionInterfaceDemo{
7  
8    public static void Main(){
9      ICollection<Animal> lst = new Collection<Animal>();
10 
11     // Add elements to the end of the empty list:
12     lst.Add(new Animal("Cat"));  lst.Add(new Animal("Dog", Sex.Female));
```

```
    lst.Add(new Animal("Mouse"));  lst.Add(new Animal("Rat"));
    lst.Add(new Animal("Mouse", Sex.Female));  lst.Add(new Animal("Rat"));
    lst.Add(new Animal("Herring", AnimalGroup.Fish, Sex.Female));
    lst.Add(new Animal("Eagle", AnimalGroup.Bird, Sex.Male));

    // Report in various ways on the animal collection:
    Print("Initial List", lst);
    ReportFemaleMale(lst);
    ReportGroup(lst);
  }

  public static void Print<T>(string explanation, ICollection<T> list){
    Console.WriteLine(explanation);
    foreach(T el in list)
      Console.WriteLine("{0, 3}", el);
    Console.WriteLine(); Console.WriteLine();
  }

  public static void ReportFemaleMale(ICollection<Animal> list){
    int numberOfMales = 0,
        numberOfFemales = 0;

    foreach(Animal a in list)
      if (a.Sex == Sex.Male) numberOfMales++;
      else if (a.Sex == Sex.Female) numberOfFemales++;

    Console.WriteLine("Males: {0}, Females: {1}",
                       numberOfMales, numberOfFemales);
  }

  public static void ReportGroup(ICollection<Animal> list){
    int numberOfMammals = 0,
        numberOfBirds = 0,
        numberOfFish = 0;

    foreach(Animal a in list)
      if (a.Group == AnimalGroup.Mammal) numberOfMammals++;
      else if (a.Group == AnimalGroup.Bird) numberOfBirds++;
      else if (a.Group == AnimalGroup.Fish) numberOfFish++;

    Console.WriteLine("Mammals: {0}, Birds: {1}, Fish: {2}",
                       numberOfMammals, numberOfBirds, numberOfFish);
  }

}
```

Program 45.23 *A program based on* **`ICollection<Animal>`** *- with a* **`Collection<Animal>`**.

On the accompanying slide we show versions of Program 45.23, which are tightly bound to the class `Collection<Animal>`, and we show a version in which we have replaced `Collection<Animal>` with `List<Animal>`.

45.19. References

[Big-O] Wikipedia: Big O Notation
http://en.wikipedia.org/wiki/Big_O_notation

46. Generic Dictionaries in C#

In the same style as our coverage of lists in Chapter 45 we will in this chapter discuss generic interfaces and classes for *dictionaries*. This covers the high-level concept of *associative arrays* and the low-level concept of *hash tables*.

46.1. Overview of Generic Dictionaries in C#

Lecture 12 - slide 24

A dictionary is a data structure that maps keys to values. A given key can have at most one value in the dictionary. In other words, the key of a key-value pair must be unique in the dictionary. A given value can be associated with many different keys.

At the conceptual level, a dictionary can be understood as an associative array (see Section 19.2) or as a collection of key-value pairs. In principle the collection classes from Chapter 45 can be used as an underlying representation. It is, however, convenient to provide a specialized interface to dictionaries which sets them apart from collections in general. In addition we often need good performance (fast lookup), and therefore it is more than justified to have special support for dictionaries in the C# libraries.

Figure 46.1 gives an overview of the generic interfaces and the generic classes of dictionaries. The figure is comparable with Figure 45.1 for collections. As such, the white boxes represent interfaces and the grey boxes represent classes. As it appears from Figure 46.1 we model dictionaries as `IEnumerables` (see Section 45.2) and `ICollections` (see Section 45.3) at the highest levels of abstractions. From the figure we can directly read that a dictionary ***is a*** `ICollection` of `KeyValuePairs`. (The **is a** relation is discussed in Section 25.2).

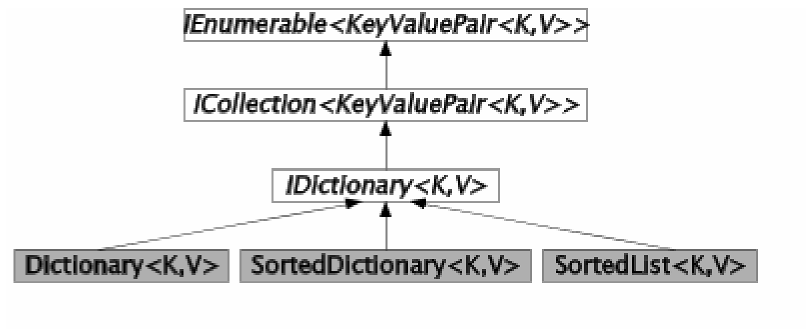

Figure 46.1 *The class and interface inheritance tree related to Dictionaries*

The symbol `K` stands for the type of keys, and the symbol `V` stands for the type of values. `KeyValuePair<K,V>` is a simple struct that aggregates a key and a value to a single object.

`Dictinonary<K,V>` is implemented in terms of a hashtable that maps objects of type `K` to objects of type `V`. `SortedDictinonary<K,V>` relies on binary search trees. `SortedList<K,V>` is based on a sorted arrays. More details can be found in Section 46.5. In Section 46.6 we review the time complexities of the operations of the three dictionary classes shown above.

46.2. The interface IDictionary<K,V>

Lecture 12 - slide 25

From Figure 46.1 we see that the interface `IDictionary<K,V>` is a subinterface of `ICollection<KeyValuePair<K,V>>`. We gave an overview of the generic interface `ICollection<T>` in Section 45.3. Because of this subinterface relationships we know that it is possible to use the operations `Contains`, `Add`, `Remove` on objects of type `KeyValuePair<K,V>`. Notice, however, that these operations are rather inconvenient because the generic class `KeyValuePair` is involved. Instead of `Add(new KeyValuePair(k,v))` we prefer another overload of `Add`, namely `Add(k,v)`. The mentioned operations `Contains`, `Add`, and `Remove` on `KeyValuePairs` are available in the `Dictionary` classes of Figure 46.1, but they are degraded to explicit interface implementations (see Section 31.8).

The following provides an overview of the operations in **`IDictionary<K,V>`**:

- *The operations prescribed in* **`ICollection<KeyValuePair<K,V>>`**
- *The operations prescribed in* **`IEnumerable<KeyValuePair<K,V>>`**
- V **`this`**[K key] - both getter and setter; the setter adds or mutates
- void **`Add`**(K key, V value) - only possible if key is not already present
- bool **`Remove`**(K key)
- bool **`ContainsKey`**(K key)
- bool **`TryGetValue`**(K key, out V value)
- `ICollection<K>`**`Keys`** - getter
- `ICollection<V>`**`Values`** - getter

`V this[K key]` is an indexer via which we can set and get a value of a given key by means of array notation (see Section 19.1). If `dict` is declared of type `IDictionary<K,V>` then the indexer notation allows us to express

```
valVar = dict[someKey];
dict[someKey] = someValue;
```

The first line accesses (gets/reads) the value associated with `someKey`. If no value is associated with `someKey` an `KeyNotFoundException` is thrown. The second line adds (sets/writes) an association between `someKey` and `someValue` to `dict`. If the association is already in the dictionary, the setter mutates the value associated with `someKey`.

The operation `Add(key,value)` adds an association between `key` and `value` to the dictionary. If the key is already associated with (another) value in the dictionary an `ArgumentException` will be thrown.

`Remove(key)` removes the association of `key` and its associated value. Via the value returned, the `Remove` operation signals if the removal was successful. `Remove` returns *false* if key is not present in the dictionary.

`ContainsKey(key)` tells if `key` is present in the dictionary.

The operation call `TryGetValue(key, valueVar)` accesses the value of `key`, and it passes the value via an output parameter (see Section 20.7). If no value is associated with key, the default value of type `V` (see Section 12.3) is passed back in the output parameter. This method is added of convenience. Alternatively, the indexer can be used in combination with `ContainsKey`.

The properties `Keys` and `Values` return collections of the keys and the values of a dictionary.

46.3. Overview of the class Dictionary<K,V>

Lecture 12 - slide 26

The generic class `Dictionary<K,V>` is based on hashtables. `Dictionary<K,V>` implements the interface `IDictionary<K,V>` as described in Section 46.2. Almost all methods and properties of `Dictionary<K,V>` are prescribed by the direct and indirect interfaces of the class. In the web version of the material we enumerate the most important operations of `Dictionary<K,V>`.

As it appears from the discussion of dictionaries above, it is necessary that two keys can be compared for equality. The equality comparison can be provided in several different ways. It is possible to pass an `EqualityComparer` object to the `Dictionary` constructor. Alternatively, we fall back on the *default equality comparer* of the key type *K*. The property `Comparer` of class `Dictionary<K,V>` returns the comparer used for key comparison in the current dictionary. See also the discussion of equality comparison in Section 42.9.

As already mentioned, a dictionary is implemented as a hash table. A hash table provides very fast access to the a value of a given key. Under normal circumstances - and with a good hash function - the run times of the access operations are constant (the run times do not depend on the size of the dictionary). Thus, the time complexity is O(1). Please consult Section 46.6 for more details on the efficiency of the dictionary operations.

46.4. Sample use of class Dictionary<K,V>

Lecture 12 - slide 27

In this section we will illustrate the use of dictionaries with a simple example. We go for a dictionary that maps objects of type `Person` to objects of type `BankAccount`. Given a `Person` object (the key) we wish to have efficient access to the person's `BankAccount` (the value).

The class `Person` is similar to Program 20.3. The class `BankAccount` is similar to Program 25.1. The exact versions of `Person` and `BankAccount`, as used in the dictionary example, can be accessed via the accompanying slide page, or via the program index of this lecture.

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

class DictionaryDemo{

  public static void Main(){

    IDictionary<Person, BankAccount> bankMap =
      new Dictionary<Person,BankAccount>(new PersonComparer());

    // Make bank accounts and person objects
    BankAccount ba1 =  new BankAccount("Kurt", 0.01),
                ba2 =  new BankAccount("Maria", 0.02),
                ba3 =  new BankAccount("Francoi", 0.03),
                ba4 =  new BankAccount("Unknown", 0.04);

    Person p1 = new Person("Kurt"),
            p2 = new Person("Maria"),
            p3 = new Person("Francoi");

    ba1.Deposit(100); ba2.Deposit(200); ba3.Deposit(300);

    // Populate the bankMap:
    bankMap.Add(p1, ba1);
    bankMap.Add(p2, ba2);
    bankMap.Add(p3, ba3);
    ReportDictionary("Initial map", bankMap);

    // Print Kurt's entry in the map:
    Console.WriteLine("{0}\n", bankMap[p1]);

    // Mutate Kurt's entry in the map
    bankMap[p1] = ba4;
    ReportDictionary("bankMap[p1] = ba4;", bankMap);

    // Mutate Maria's entry in the map. PersonComparer crucial!
    ba4.Deposit(400);
    bankMap[new Person("Maria")] = ba4;
    ReportDictionary("ba4.Deposit(400);  bankMap[new Person(\"Maria\")] = ba4;",
bankMap);

    // Add p3 yet another time to the map
    // Run-time error: An item with the same key has already been added.
/*  bankMap.Add(p3, ba1);
    ReportDictionary("bankMap.Add(p3, ba1);", bankMap);
 */

    // Try getting values of some given keys
    BankAccount ba1Res = null,
                ba2Res = null;
    bool res1 = false,
         res2 = false;
    res1 = bankMap.TryGetValue(p2, out ba1Res);
    res2 = bankMap.TryGetValue(new Person("Anders"), out ba2Res);
    Console.WriteLine("Account: {0}. Boolean result {1}", ba1Res, res1);
    Console.WriteLine("Account: {0}. Boolean result {1}", ba2Res, res2);
    Console.WriteLine();

    // Remove an entry from the map
    bankMap.Remove(p1);
    ReportDictionary("bankMap.Remove(p1);", bankMap);

    // Remove another entry - works because of PersonComparer
    bankMap.Remove(new Person("Francoi"));
    ReportDictionary("bankMap.Remove(new Person(\"Francoi\"));", bankMap);
  }

  public static void ReportDictionary<K, V>(string explanation,
                                            IDictionary<K,V> dict){
    Console.WriteLine(explanation);
    foreach(KeyValuePair<K,V> kvp in dict)
      Console.WriteLine("{0}: {1}", kvp.Key, kvp.Value);
    Console.WriteLine();
  }
}

public class PersonComparer: IEqualityComparer<Person>{

  public bool Equals(Person p1, Person p2){
    return (p1.Name == p2.Name);
  }

```

```
83    public int GetHashCode(Person p){
84      return p.Name.GetHashCode();
85    }
  }
```

Program 46.1 *A program working with* ***`Dictionary<Person,BankAccount>`***.

In line 8-9 we make the dictionary `bankMap` of type `Dictionary<Person,BankAccount>`. We pass an instance of class `PersonComparer`, see line 76-86, which implements `IEqualityComparer<Person>`. In line 11-19 we make sample `BankAccount` and `Person` objects, and in line 24-26 we populate the dictionary `bankMap`.

In line 30 we see how to access the bank account of person `p1` (Kurt). We use the provided indexer of the dictionary. In line 33 we mutate the `bankMap`: Kurt's bank account is changed from the one referenced by `ba1` to the one referenced by `ba4`. In line 38 we mutate Maria's bank account in a similar way. Notice, however, that that the relative weak equality of `Person` objects (implemented in class `PersonComparer`) implies that the new `person("Maria")` in line 38 is equal to the person referenced by `p2`, and therefore line 38 mutates the dictionary entry for Maria.

In line 43 we attempt add yet another entry for Francoi. This is illegal because there is already an entry for Francoi in the dictionary. If the comments around line 43 are removed, a run time error will occur.

In line 52-53 we illustrate `TryGetValue`. First, in line 52, we attempt to access Maria's account. The out parameter `baRes1` is assigned to Maria's account and `true` is returned from the method. In line 53 we attempt to access the account of a brand new `Person` object, which has no bank account in the dictionary. `null` is returned through `ba2Res`, and `false` is returned from the method.

Finally, in line 58-64 we remove entries from the dictionary by use of the `Remove` method. First Kurt's entry is removed after which Francoi's entry is removed.

The output of the program is shown in Listing 46.2 (only on web).

Exercise 12.3. *Switching from Dictionary to SortedDictionary*

The program on this slide instantiates a **`Dictionary<Person,BankAccount>`**. As recommended earlier in this lecture, we should work with the dictionary via a variable of the interface type **`IDictionary<K,V>`**.

You are now asked to replace **`Dictionary<Person,BankAccount>`** with **`SortedDictionary<Person,BankAccount>`** in the above mentioned program.

This causes a minor problem. Identify the problem, and fix it.

Can you tell the difference between the output of the program on this slide and the output of your revised program?

You can access the `BankAccount` and `Person` classes in the web version of the material.

46.5. Notes about Dictionary Classes

Lecture 12 - slide 28

As can be seen from Figure 46.1 several different generic classes implement the `IDictionary<K,V>` interface. `Dictionary<K,V>`, as discussed in Section 46.3 and Section 46.4 is based on a hash table representation. `SortedDictionary<K,V>` is based on a binary tree, and (as the name signals) `SortedList<K,V>` is based on an array of key/value pairs, sorted by keys.

The following provides an itemized overview of the three generic dictionary classes.

- Class **`Dictionary<K,V>`**
  - Based on a hash table
  - Requires that the keys in type **`K`** can be compared by an **`Equals`** operation
  - Key values should not be mutated
  - The efficiency of class dictionary relies on a good hash function for the key type **`K`**
    - Consider overriding the method **`GetHashCode`** in class `K`
  - A dictionary is enumerated in terms of the struct **`KeyValuePair<K,V>`**
- Class **`SortedDictionary<K,V>`**
  - Based on a binary search tree
  - Requires an **`IComparer`** for keys of type **`K`** - for ordering purposes
    - Provided when a sorted dictionary is constructed
- Class **`SortedList<K,V>`**
  - Based on a sorted collection of key/value pairs
    - A resizeable array
  - Requires an **`IComparer`** for keys, just like **`SortedDictionary<K,V>`**.
  - Requires less memory than **`SortedDictionary<K,V>`**.

When you have to chose between the three dictionary classes the most important concern is the different run time characteristics of the operations of the classes. The next section provides an overview of these.

46.6. Time complexity overview: Dictionary classes

Lecture 12 - slide 29

We will now review the time complexities of the most important dictionary operations. This is done in the same way as we did for collections (lists) in Section 45.17. We will assume that we work on a dictionary that holds *n* entries of key/value pairs.

Operation	**`Dictionary<K,V>`**	**`SortedDictionary<K,V>`**	**`SortedList<K,V>`**
`this[key]`	*O(1)*	*O(log n)*	*O(log n) or O(n)*
`Add(key,value)`	*O(1) or O(n)*	*O(log n)*	*O(n)*
`Remove(key)`	*O(1)*	*O(log n)*	*O(n)*
`ContainsKey(key)`	*O(1)*	*O(log n)*	*O(log n)*
`ContainsValue(value)`	*O(n)*	*O(n)*	*O(n)*

Table 46.1 *Time complexities of important operations in the classes `Dictionary<K,V>`, `SortedDictionary<K,V>`, and `SortedList<K,V>`.*

As noticed in Section 46.5 an object of type `Dictionary<K,V>` is based on hash tables. Eventually, it will be necessary to enlarge the hashtable to hold new elements. It is good wisdom to enlarge the hashtable when it becomes half full. The *O(1) or O(n)* time complexity for `Add` reflects that a work proportional to *n* is needed when it becomes necessary to enlarge the hash table.

Most operations on the binary tree representation of `SortedDictionary<K,V>` are logarithmic in *n*. The only exception (among the operations listed in the table) is `ContainsValue`, which in the worst case requires a full tree traversal.

In `SortedList<K,V>` the indexer is efficient, *O(log n)* when an existing item is mutated. If use of the indexer causes addition of a new entry, the run time is the same as the run time of `Add`. Adding elements to a sorted list requires, in average, that half of the elements are pushed towards the end of the list in order to create free space for the new entry. This is an *O(n)* operation. `Remove` is symmetric, pulling elements towards the beginning of the list, and therefore also *O(n)*. `ContainsKey` is efficient because we can do binary search on the sorted list. `ContainsValue` requires linear search, and therefore it is an *O(n)* operation.

Given the table in Table 46.1 it is tempting to conclude that `Dictionary<K,V>` is the best of the three classes. Notice, however, that the difference between a constant run time *c1* and *c2 log(n)* is not necessarily significant. If the constant *c1* is large and the constant *c2* is small, the binary tree may be an attractive alternative. Furthermore, we know that the hashtable will be slow when it is almost full. In that case more and more collisions can be expected. At some point in time the hash table will stop working if it is not resized. This is not an issue if we work with balanced binary trees. Finally, the hashtable depends critically on a good hash function, preferable programmed specifically for the key type `K`. This is not an issue if we use binary trees.

47. Non-generic Collections in C#

This is a short chapter in which we discuss the non-generic collection classes. You may encounter use of these classes in many older C# programs. In Section 44.1 these collection classes were called *first generation collection classes*.

47.1. The non-generic collection library in C#

Lecture 12 - slide 31

The overview of the non-generic collection interfaces and classes in Figure 47.1 is a counterpart to the sum of Figure 45.1 and Figure 46.1. The white boxes represent interfaces and the grey boxes represent classes. Most classes and interfaces shown in Figure 46.1 belong to the namespace `System.Collections`.

The non-generic collection classes store data of type **`Object`**

As the most important characteristics, the elements of the lists are of type `Object`. Both keys and values of dictionaries are `Objects`. Without use of type parametrization, there are no means to constraint the data in collections to of a more specific type. Thus, if we for instance work with a collection of bank accounts, we cannot statically guarantee that all elements of the collection are bank accounts. We may accidentally insert an object of another type. We will find the error at runtime. Most likely, an exception will be raised when we try to cast an `Object` to `BankAccount`.

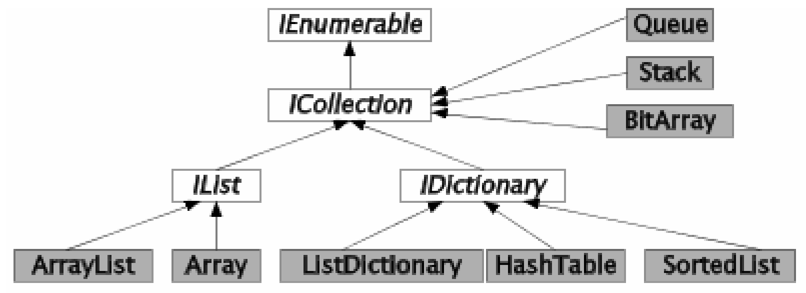

Figure 47.1 *The class and interface inheritance tree related to collections*

The `IEnumerable`, `ICollection`, `IList` and `IDictionary` interfaces of Figure 47.1 are natural counterparts to the generic interfaces `IEnumerable<T>`, `ICollection<T>`, `IList<T>` and `IDictionary<K,V>`.

The class `ArrayList` corresponds to `List<T>`. As such, `ArrayList` is a class with a rich repertoire of operations for searching, sorting, and range operations. `ArrayList` is undoubtedly the most widely used collection class in C# 1.0 programs.

The `Array` class shown next to `ArrayList` in Figure 47.1 deserves some special clarification. It belongs to the `System` namespace. You cannot instantiate class `Array` in your programs, because `Array` is an abstract class. And you cannot use `Array` as a superclass of one of your own classes. So, class `Array` seems pretty useless. At least it is fair to state the class `Array` is rather special compared to the other classes in Figure 47.1.

Let us now explain the role of class `Array`. As mentioned earlier, see Section 28.2 , class `Array` acts as the superclass of all "native" array types in C#. (See the discussion of arrays in Section 6.4). Consequently, all the nice operation in `System.Array` can be used on all "native" arrays that you use in your C# programs. If, for instance, we have the array declarations

```
int[] ia = new int[3];
string[] sa = new string[5,6];
BankAccount[] baa = new BankAccount[10];
```

the following are legal expressions

```
ia.Length
a.Rank
Array.BinarySearch(ia, 5)
Array.Find(sa, IsPalindrome)
Array.Sort(baa)
```

In the `Array` class, you should pay attention to the (overloaded) static method `CreateInstance`, which allows for programmatic creation on an arbitrary array. The `Array` instance methods `GetValue` and `SetValue` allow us to access elements in arbitrary arrays - independent of element type and rank.

When we talk about "native arrays" in C# we refer to the array concept implemented in the language as such. The compiler provides special support for these native arrays. In contrast, generic and non-generic collections are provided via the class library. The C# compiler and the C# interpreter do not have particular knowledge or support of the collection classes. We could have written these classes ourselves! It is interesting to notice that the native arrays, as derived from class `Array` in Figure 47.1, are type safe. The type safeness of native arrays is due to the special support by the compiler, which allows for declaration of the element types of the arrays (see the examples of `int`, `string`, and `BankAccount` arrays above).

The class `HashTable` in Figure 47.1 corresponds to the generic class `Dictionary<K,V>`, see Section 46.3 and Section 46.4).

The class `ListDictionary`, which belongs to the namespace `System.Collections.Specialized`, has no natural generic counterpart. `ListDictionary` is based on linear search in an unordered collection of key/value pairs. `ListDictionary` should therefore only be used for small dictionaries.

As the name suggests, class `SortedList` corresponds to `SortedList<K,V>`. Both rely on a (linear) list representation, sorted by keys.

The class `BitArray` is - by nature - a non-generic collection class. The binary digit 1 is represented as boolean *true*, and the binary digit 0 is represented as boolean *false*. `BitArray` provides a compact representation of a bit arrays. In the context of indexers, see Program 19.4, we have earlier discussed a partial reproduction of the class `BitArray`.

In addition to the types shown in Figure 47.1 there exist some specialized collections in the namespace `System.Collections.Specialized`. As an example, the class `StringCollection` is a collection of strings. The class `CollectionBase` in the namespace `System.Collection` is intended as the superclass of new, specialized collection classes. In the documentation of this class, an example shows how to define an `Int16Collection` as a subclass of `CollectionBase`. Needless to say, *all these classes are obsolete* relative to both C#2.0 and C#3.0. As of today, the classes may be necessary for backward compatibility, but, unfortunately, they also add to the complexity of the .NET class libraries.

48. Patterns and Techniques

In earlier parts of this material (Section 31.6 and Section 45.2) we have at length discussed enumerators in C#, including their relationship to **foreach** loops.

In this section we first briefly rephrase this to the design pattern known as ***Iterator***. Following that we will show how to implement iterators (enumerators) with use of **yield return**, which is a variant of the **return** statement.

48.1. The Iterator Design Pattern

Lecture 12 - slide 34

The ***Iterator*** design pattern provides sequential access to an aggregated collection. At an overall level, an iterator

- Provides for a smaller interface of the collection class
  - All members associated with traversals have been refactored to the iterator class
- Makes it possible to have several simultaneous traversals
- Does not reveal the internal representation of the collection

As we have seen in Section 31.6 and Section 45.2, traversal of a collection requires a few related operations, such as `Current`, `MoveNext`, and `Reset`. We could imagine a slightly more advanced iterator which could move backwards as well. With use of iterators we have factored these operations out of the collection classes, and organized them in iterators (enumerators). With this refactoring, a collection can be asked to deliver an iterator:

```
  aCollection.GetEnumerator()
```

Each iterator maintains the state, which is necessary to carry out a traversal of a collection. If we need two independent, simultaneous traversals we can ask for two iterators of the collections. This could, for instance be used to manage simultaneous iteration from both ends of a list.

In more primitive collections, such as linked lists (see Section 45.14) it is necessary to reveal the object structure that keeps the list together. (In `LinkedList<T>` this relates to the details of `LinkedListNode<T>` instances). With use of iterators it is not necessary to reveal such details. An iterator is an encapsulated, abstract representation of some state that manages a traversal. The concrete representation of this state is not leaked to clients. This is very satisfactory in an object-oriented programming context.

Iterators (enumerators) are typically used via foreach loops. As an alternative, it is of course also possible to use the operations in the `IEnumerator` interface directly to carry out traversals. Exercise 12.4 is a opportunity to train such a more direct use of iterators.

Exercise 12.4. *Explicit use of iterator - instead of using foreach*

In this program we will make direct use of an iterator (an enumerator) instead of traversing with use of foreach.

In the animal collection program, which we have seen earlier in this lecture, we traverse the animal collections several times with use of foreach. Replace each use of foreach with an application of an iterator.

48.2. Making iterators with yield return

Lecture 12 - slide 35

In this section we will show how to use the special-purpose **yield return** statement to define iterators, or as they are called in C#, enumerators. First, we will program a very simple collection of up to three, fixed values. Next we will revisit the integer sequence enumeration, which can be found in Section 58.3.

In Program 48.1 we will program a collection class, called `GivenCollection`, which just covers zero, one, two or three values of some arbitrary type `T`. As a simpleminded approach, we represent these `T` values with three instance variables of type `T`, and with three boolean variables which tells if the corresponding `T` values are present. As an invariant, the instance variables are filled from the lower end. It would be tempting to use the type `T?` instead of `T`, and the value `null` for a missing value. But this is not possible if `T` is class.

It is important that the class `GivenCollection` implements the generic interface **`IEnumerable<T>`**. Because this interface, in turn, implements the non-generic `IEnumerable`, we must both define the generic and the non-generic `GetEnumerator` method. The latter must be defined as an explicit interface (see Section 31.8), in order not to conflict with the former. If we forget the non-generic `GetEnumerator`, we get a slightly misleading error message:

> '**`GivenCollection<T>`**' does not implement interface member 'System.Collections.IEnumerable.GetEnumerator()'.
> '**`GivenCollection<T>`**' is either static, not public, or has the wrong return type.

This message can cause a lot of headache, because the real problem (the missing, non-generic `GetEnumerator` method) is slightly camouflaged in the error message.

The implementation of the non-generic enumerator just delegates its work to the generic version.

The implementation of the generic `Enumerator` method uses the **yield return** statement. Let us assume that an instance of `GivenCollection<T>` holds three `T` values (in `first`, `second`, and `third`). The three boolean variables `firstDefined`, `secondDefined`, and `thirdDefined` are all true. The `GetEnumerator` method has three yield return statements in sequence (see line 50-52). By means of these, `GetEnumerator` can return three values before it is done. This is entirely different from a normal method, which only returns once (after which it is done). The `GetEnumerator` in class `GivenCollection` acts as a coroutine in relation to its calling place (which is the **foreach** statement in the client program Program 48.2). A coroutine can resume execution at the place where execution stopped in an earlier call. A normal method always (re)starts from its first statement each time it is called.

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

public class GivenCollection<T> : IEnumerable<T>{

  private T first, second, third;
  private bool firstDefined, secondDefined, thirdDefined;

  public GivenCollection(){
    this.firstDefined = false;
    this.secondDefined = false;
    this.thirdDefined = false;
  }

  public GivenCollection(T first){
    this.first = first;
    this.firstDefined = true;
    this.secondDefined = false;
    this.thirdDefined = false;
  }

  public GivenCollection(T first, T second){
    this.first = first;
    this.second = second;
    this.firstDefined = true;
    this.secondDefined = true;
    this.thirdDefined = false;
  }

  public GivenCollection(T first, T second, T third){
    this.first = first;
    this.second = second;
    this.third = third;
    this.firstDefined = true;
    this.secondDefined = true;
    this.thirdDefined = true;
  }

  public int Count(){
    int res;
    if (!firstDefined) res = 0;
    else if (!secondDefined) res = 1;
    else if (!thirdDefined) res = 2;
    else res = 3;
    return res;
  }

  public IEnumerator<T> GetEnumerator(){
    if (firstDefined) yield return first;
    if (secondDefined) yield return second;  // not else
    if (thirdDefined) yield return third;    // not else
  }

  IEnumerator IEnumerable.GetEnumerator(){
    return GetEnumerator();
  }

}
```

Program 48.1 *A collection of up to three instance variables of type T - with an iterator.*

In Program 48.2 we show a simple program that instantiates a `GivenCollection` of the integers 7, 5, and 3. The **foreach** loop in line 11-12 traverses the three corresponding instance variables, and prints each of them.

```
1  using System;
2  
3  class Client{
4  
5    public static void Main(){
6  
7       GivenCollection<int> gc = new GivenCollection<int>(7,5,3);
8  
9       Console.WriteLine("Number of elements in givenCollection: {0}",
10                        gc.Count());
11      foreach(int i in gc){      // Output:  7 5 3
12        Console.WriteLine(i);
13      }
14  
15   }
16  
17 }
```

Program 48.2 *A sample iteration of the three instance variable collection.*

Exercise 12.5. *The iterator behind a yield*

Reprogram the iterator in class `GivenCollection` without using the **yield return** statement in the `GetEnumerator` method.

Let us now revisit the integer enumeration classes of Section 58.3. The main point in our first discussion of these classes was the ***Composite*** design pattern, cf. Section 32.1, as illustrated in Figure 58.1 of Section 58.3. The three classes `IntInterval`, `IntSingular`, and `IntCompSeq` all inherit the abstract class `IntSequece`. You can examine the abstract class `IntSequence` in Program 58.9 in the appendix of this material. The three concrete subclasses were programmed in Program 58.10, Program 58.11, and Program 58.12.

The `GetEnumerator` methods of `IntInterval`, `IntSingular`, and `IntCompSeq` are all emphasized below in Program 48.3, Program 48.4, and Program 48.5. Notice the use of **yield return** in all of them.

In Program 48.3 the if-else of `GetEnumerator` in line 19-24 distinguishes between increasing and decreasing intervals. The `GetEnumerator` method of `IntSingular` is trivial. The `GetEnumerator` method of `IntCompSeq` in Program 48.5 is surprisingly simple - at least compared with the counterpart in Program 58.12. The two foreach statements (in sequence) in line 19-22 activate all the machinery, which we programmed manually in Program 58.12. This includes recursive access to enumerators of composite sequences.

The simplicity of enumerators, programmed with yield return, is noteworthy compared to all the underlying stuff of explicitly programmed classes that implement the interface `IEnumerator`.

Iterators (iterator blocks), programmed with **`yield return`**, are only allowed to appear in methods that implement an enumerator or an enumerable interface (such as `IEnumerator` or `IEnumerator` and their generic counterparts). Such methods are handled in a very special way by the compiler, and a number of restrictions apply to these methods. The compiler generates all the machinery, which we program ourselves when a class implements the enumerator or enumerable interfaces. Methods with iterator blocks that implement and enumerator or an enumerable interface return an enumerator object, on which the `MoveNext` can be called a number of times. For more details on iterators please consult Section 10.14 in the C# 3.0 Language Specification [csharp-3-spec].

```
public class IntInterval: IntSequence{

  private int from, to;

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

  public override int? Min{
    get {return Math.Min(from,to);}
  }

  public override int? Max{
    get {return Math.Max(from,to);}
  }

  public override IEnumerator GetEnumerator (){
    if (from < to)
     for(int i = from; i <= to; i++)
       yield return i;
    else
     for(int i = from; i >= to; i--)
       yield return i;
  }

}
```

Program 48.3 *The class IntInterval - Revisited.*

```
public class IntSingular: IntSequence{

  private int it;

  public IntSingular(int it){
    this.it = it;
  }

  public override int? Min{
    get {return it;}
  }

  public override int? Max{
    get {return it;}
  }

  public override IEnumerator GetEnumerator(){
    yield return it;
  }
}
```

Program 48.4 *The class IntSingular - Revisited.*

```
public class IntCompSeq: IntSequence{

  private IntSequence s1, s2;

  public IntCompSeq(IntSequence s1, IntSequence s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  public override int? Min{
    get {return (s1.Min < s2.Min) ? s1.Min : s2.Min;}
  }

  public override int? Max{
    get {return (s1.Max > s2.Max) ? s1.Max : s2.Max;}
  }

  public override IEnumerator GetEnumerator (){
    foreach(int i in s1)
      yield return i;
    foreach(int i in s2)
      yield return i;
  }

}
```

Program 48.5 *The class IntCompSeq - Revisited.*

In the web edition of the material we show a sample client program that contains a couple of `IntSequences`.

48.3. References

[Csharp-3-spec] "The C# Language Specification 3.0",

49. Correctness

This is the first chapter in the lecture about contracts and assertions. We all want to write correct programs. But what is correctness? Program correctness is always relative to something else. In this lecture we will discuss program correctness relative to a *program specification*. In Chapter 50, (the next chapter) we will take a closer look at a particular approach to program specification, on which the rest of this lecture will be based.

49.1. Software Qualities

Lecture 13 - slide 2

Program correctness is one of several *program qualities*. A software quality is a positive property of program. There are many different software qualities that may be considered and promoted. In Table 49.1 we list a number of important program qualities.

Quality	**Description**	**Contrast**
Correct	Satisfies expectations, intentions, or requirements	Erroneous
Robust	Can resist unexpected events	Fragile
Reusable	Can be used in several contexts	Application specific
Simple	Avoids complicated solutions	Complex
Testable	Constructed to ease revelation of errors	-
Understandable	Mental manageability	Cryptic

Table 49.1 *Different program qualities listed by name, description, and (for selected qualities) a contrasting, opposite quality*

Of all software qualities, correctness play a particular important role. Program correctness is in a league of its own. Who would care about robustness, reusability, and simplicity of an incorrect program?

49.2. Correctness

Lecture 13 - slide 3

Software correctness is only rarely an absolute concept. Correctness should be seen relative to something else. We will distinguish between program correctness relative to

- The programmers own, immediate comprehension
  - Not formulated - not documented - volatile - easily forgotten
  - Sometimes incomplete
- A program specification
  - Formulated - written
  - Well-considered and agreed upon
  - Formal or informal
  - Part of the program

At the time the program is written, it may be tempting to rely on the comprehension and specification in the mind of the programmer. It is not difficult to understand, however, that such a specification is volatile. The specification may slide away from the original understanding, or it may totally fade away. In a software house it may also easily be the case that the programmer is replaced. Of these reasons it is attractive to base correctness on written and formal specifications.

In the following section we will discuss written and formal specifications that are based on mathematical grounds.

49.3. Specifications

Lecture 13 - slide 4

We will introduce the following straightforward definition of a specification:

A *program specification* is a definition of what a computer program is expected to do [Wikipedia].

What - *not how*.

Notice that specifications answer *what questions*, not *how questions*.

In the area of formal mathematically-oriented specifications, the following two variants are well-known:

- **Algebraic specifications**
  - Equations that define how certain operations work on designated constructors
- **Axiomatic specifications**
  - Logical expressions - assertions - associated with classes and operations
  - Often divided into invariants, preconditions, and postconditions

We will first study an algebraic specification of a stack, see Program 49.1. We have already encountered this specification earlier in the material, namely in the context of our discussion of abstract data types in Section 1.5. From line 4-11 we declare the syntax of the operations that work on stacks. The operations are categorized as constructors, destructors, and selectors. As the name suggests, constructors are operations that constructs a stack. Both `push` and `pop` are *functions* that return a stack. This is different from the imperative stack *procedures* we experienced in Program 30.1, which mutate the stack without returning any value.

An arbitrary stack can be constructed in terms of one or more constructors. Destructors are operations that work on stacks. (The term "destructor" may be slightly misleading). Any stack can be constructed without use of destructors. As an example, the expression `pop(push(5, pop (push (6, push (7, new ()))))))` is equivalent with `push(7, new ())`. The selectors extract information about the stack.

```
1  Type stack [int]
2    declare
3          constructors
4                   new () -> stack;
5                   push (int, stack) -> stack;
6          destructors
7                   pop (stack) -> stack;
8          selectors
9                   top (stack) -> int;
10                  isnew (stack) -> bool;
11   for all
12         i in int;
13         s in stack;
14   let
15         pop (new()) = error;
16         pop (push (i,s)) = s;
17         top (new()) = error;
18         top (push (i,s)) = i;
19         isnew (new()) = true;
20         isnew (push(i,s)) = false;
21   end
22 end stack.
```

Program 49.1 *An algebraic specification of a stack.*

The lines 12-21 define the *meaning* (also known as the *semantics*) of the stack. It tells us what the concept of a stack is all about. The idea is to define equations that express how each destructor and each selector work on expressions formulated in terms of constructors. The equation in line 16 specifies that it is an error to pop the empty stack. The equation in line 17 specifies that pop applied on stack `s` on which we have just pushed the integer `i` is equivalent with `s`. Please consider the remaining equations and make sure that you understand their meaning relative to your intuition of the stack concept.

The specification in Program 49.1 tells us what a stack is. It is noteworthy that the specification in Program 49.1 defines the stack concept without any binding to a concrete representation of a stack. The specification gives very little input to the programmer about how to implement a stack with use of a list or an array, for instance. A good specification answers *what questions*, not *how questions*.

If you wish to see other similar specifications of abstract datatypes, you may review our specifications of natural numbers and booleans in Program 1.9 and Program 1.10 respectively.

Below, in Program 49.2 we show an axiomatic specification of a single function, namely the square root function. An axiomatic specification is formulated in terms of a precondition and a postcondition. The precondition specifies the prerequisite for activation of the square root function. It states that it is only possible to calculate the square root of non-negative numbers. The precondition constrains the output of the function. In case of the square root function, the square of the result should be very close to the input.

```
sqrt(x: Real) -> Real

  precondition: x >= 0;

  postcondition: abs(result * result - x) <= 0.000001
```

Program 49.2 *An axiomatic specification of the squareroot function.*

In the rest of this lecture we will study object-oriented programming, in which methods can be specified with preconditions and postconditions.

50. Specification with preconditions and postconditions

As exemplified at the end of the previous chapter, preconditions and postconditions can be used to specify the meaning of a function. In this chapter we will study preconditions and postconditions in more details.

50.1. Logical expressions

Lecture 13 - slide 6

Logical expressions and assertions form the basis of preconditions and postconditions. Consequently, we define the concepts of logical expressions and assertions before preconditions and postconditions:

A *logical expression* is an expression of type boolean

An *assertion* is a logical expression, which, if false, indicates an error [Foldoc]

A *precondition* of an operation is an assertion which must be true just before the operation is called

A *postcondition* of an operation is an assertion which must be true just after the operation has been completed

We have worked with logical expressions numerous times during this course. Logical expressions are formed by relational, equational, conjunctional (and) and disjunctional (or) operators. You find these operators at level 3, 4, 8, and 9 in Table 6.1.

Assertions are also used in the context of program testing. In Section 55.7 we surveyed a large collection of assertions, which are available in the NUnit testing tools for C#. As stated in Section 55.8, an assertion in a test case, which returns the value false, causes a failure. A failed testcase signals that the unit under test is incorrect. Assertions used in test cases are similar to assertions found in postconditions.

We can now characterize a precondition in the following way:

- A precondition states if it makes sense to call an operation
- The precondition is a *prerequisite* for the activation

The precondition is typically formulated in terms of the formal parameters of the operation.

Similarly, a postcondition can be characterized as follows:

- A postcondition states if the operation returns the desired result, or has the desired effect, relative to the given parameters that satisfy the precondition
- The postcondition defines the *meaning* of the operation

The postcondition of a procedure or function F must be fulfilled if the precondition of F holds, and if F terminates (F runs to its completion).

50.2. Examples of preconditions and postconditions

Lecture 13 - slide 7

We will now study preconditions and postconditions of the operations in a circular list. A circular list is a linked list, in the sense we discussed in Section 45.14. However, the circular list discussed in this section is only single-linked. The distinctive characteristics of a circular list are the following:

1. The last `LinkedListNode` is linked to the first `LinkedListNode` of the list
2. The `CircularList` object refers to the `LinkedListNode` of the last element instead of the `LinkedListNode` of the first element.

We show a circular list with five elements in Figure 50.1. The idea of referring the last element instead of the first element from the `CircularList` object means that both the front and the rear of the list can be reached in constant time. In many context, this is a very useful property. Notice also, that it is possible to deal with double-linked circular lists as well.

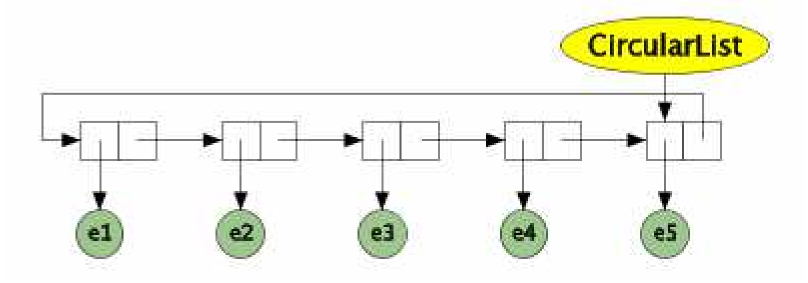

Figure 50.1 *A circular list. The large yellow object represents the circular list as such. The circular green nodes represent the elements of the list. The rectangular nodes are instances of a class akin to* `LinkedListNode`*, which connect the constituents of the list together.*

Below we specify the operations of the circular list with preconditions and postconditions. The specification in Program 50.1 defines the meaning of operations of the yellow object in Figure 50.1. In the program listing, the preconditions are marked with keyword **require**, and shown in **red**. The postconditions are marked with the keyword **ensure**, and shown in **blue**. The names of the keywords stem from the object-oriented programming language Eiffel [Meyer97, Meyer92, Switzer93], which is strong in the area of assertions. Apart from that, the syntax used in Program 50.1 is C# and Java like.

```
class CircularList {

  // Construct an empty circular list
  public CircularList()
    require true;
    ensure Empty();

  // Return my number of elements
  public int Size()
    require true;
    ensure size = CountElements() && noChange;

  // Insert el as a new first element
  public void InsertFirst(Object el)
    require !Full();
    ensure !Empty() && IsCircular() && IsFirst(el);

  // Insert el as a new last element
  public void InsertLast(Object el)
    require !Full();
    ensure !Empty() && IsCircular() && IsLast(el);

  // Delete my first element
  public void DeleteFirst()
    require !Empty();
    ensure
      Empty() ||
      (IsCircular() && IsFirst(old RetrieveSecond()));

  // Delete my last element
  public void DeleteLast()
    require !Empty();
    ensure
      Empty() ||
      (IsCircular() && isLast(old RetrieveButLast()));

  // Return the first element in the list
  Object RetrieveFirst()
    require !Empty();
    ensure IsFirst(result) && noChange;

  // Return the last element in the list
  Object RetrieveLast()
    require !Empty();
    ensure IsLast(result) && noChange;
}
```

Program 50.1 *Circular list with preconditions and postconditions.*

The precondition `true` of the constructor says that there are no particular requirements to call the constructor. This is natural and typical. The postcondition of the constructor expresses that the constructor makes an empty circular list.

The operation `Size` returns an integer corresponding to the counted number of elements in the list. `CountElements` is an operation, which counts the elements in the list. In a particular implementation of `CircularList`, the operation `Size` my return the value of a private instance variable which keeps track of the total number of elements in the list. `nochange` is a special assertion, which ensures that the state of the list has not changed due the execution of the `Size` operation. We see that the postcondition expresses the consistency between the value returned by `Size`, and the counted number of elements in the list.

The operation `InsertFirst` is supposed to insert an element, to become the first element of the list (the one shown at the left hand side of Figure 50.1). The precondition expresses that the list must not be full before the insertion. The postcondition expresses that the list is not empty after the insertion, that it is still circular, and that `el` indeed is the first element of the list. The specification of the operation `InsertLast` is similar to `InsertFirst`.

The operation `DeleteFirst` requires as a precondition a non-empty list. The postcondition of `DeleteFirst` expresses that the list either is empty or circular. If the list is non-empty (and therefore circular) after the deletion, the second element before the deletion must be the first element after the deletion. Notice the modifier `old`. The value of `Old(expression)` is the value of `expression`, as evaluated in the state just before the current operation is executed. `DeleteLast` is symmetric to `DeleteFirst`.

`RetrieveFirst` returns the first element of the list. The precondition of `RetrieveFirst` says that the list must be non-empty in order for this operation to make sense. The postcondition says that the result is indeed the first element, and that `RetrieveFirst` is a pure function (it does not mutate the state of the circular list). `RetriveLast` is symmetrical to `RetrieveFirst`.

What about the operations `Empty`, `Full`, `CountElements`, `IsCircular`, `IsFirst`, `IsLast`, `RetrieveSecond`, and `RetrieveButLast`? They are intended to be auxiliary, public boolean operations in the circular list. In an implementation of `CircularList` we must implement these operations. They are supposed to be implemented as simple as possible, and they are not supposed to carry preconditions and postconditions. In order to be operational (meaning that the specification can be confirmed at run-time) these auxiliary operations be must be implemented. Nothing comes for free! In reality, we check the consistency between the operations listed in Program 50.1 and the auxiliary boolean operations. An inconsistency reveals an error in either the circular list operations, or in the auxiliary operations. The necessary auxiliary operations are typically much simpler than the circular list operations, and therefore an inconsistency most often will reveal an error in the way we have implemented a circular list operation.

50.3. An Assertion Language

Lecture 13 - slide 8

We are now about to focus on the language in which we formulate the assertions (preconditions and postconditions). In the previous section we have studied examples, in which we have met several features in the assertion language.

As it will appear, we are pragmatic with respect to the assertion language. The reason is that we allow programmed, boolean functions to be used in the assertion language. These boolean function are siblings to the functions that we are about to specify.

It is an important goal that the preconditions and the postconditions should be checkable at program execution time. Thus, the it should be possible and realistic to evaluation the assertions at run-time.

The following items characterize the assertions language:

- Logical expressions - as in the programming language
- Programmed assertions - via boolean functions of the programming language
    - Should be simple functions
    - Problems if there are errors in these
- Universal (*"for all..."*) and existential (*"there exists..."*) quantifiers
    - Requires programming - iteration - traversal
    - It may be expensive to check assertions with quantifiers
- Informal assertions, written in natural language
    - Cannot be checked
    - Much better than nothing
- Special means of expression
    - `old Expr` - The value of the expression at the beginning of the operation
    - `nochange` - A simple way to state that the operation has not changed the state of the object

Use of universal and existential quantifiers, known from mathematical formalisms, makes it hard to check the assertions. Therefore such means of expressions do not exist directly in the assertion language. If we wish to express *for all* ... or *there exists* ... it must be programmed explicitly in boolean functions.

We may easily encounter elements of a specification that we cannot (or will no) check by programmed exceptions. It may be too expensive, or too complicated to program boolean functions which represent these elements. In such situations we may wish to fall back on informal assertions, similar to comments.

50.4. References

[Switzer93]	Robert Switzer, *Eiffel and Introduction*. Prentice Hall, 1993.
[Meyer92]	Bertrand Meyer, *Eiffel the Language*. Prentice Hall, 1992.
[Meyer97]	Bertrand Meyer, *Object-oriented software construction, second edition*. Prentice Hall, 1997.

51. Responsibilities and Contracts

This section is about responsibilities and contracts, and their connection to preconditions and postconditions. Recall from Section 2.2 in the initial lecture that we already touched on responsibilities in the slipstream of the pizza delivery example, see Figure 2.1. At the end of the chapter, in Section 51.8 we briefly discuss Design by Contract, which broadens the scope for applicability of contracts in the development process.

51.1. Division of Responsibilities

Lecture 13 - slide 10

A class encapsulates some description of state, and some operations. A subset of the operations make up the interface between the class and other classes. All together, the class manages a certain amount of *responsibility*. Internally, the class is responsible for keeping the state consistent and sound. Externally, the operations of the class are responsible for delimitation of the messages that they handle, and the quality of the work (results) the operations deliver.

It is bad if a class is irresponsible. Class irresponsibility may occur if a pair classes both expect the other class to be responsible.

It is also bad if a class is too responsible. A pair of over-responsible classes redundantly care about the same properties. This is not necessary, and it bloats the amount of program lines in the implementation of the classes.

This leads us to the essence of this and the following sections, namely division of responsibilities. Let us first enumerate the consequences of well-defined and ill-defined division of responsibilities:

- Without well-defined division of responsibilities
  - All classes accept a large responsibility
  - All program parts check all possible conditions (defensive programming)
  - *Makes a large program even larger*
- With well-defined division of responsibilities
  - Operations can safely operate under given assumptions
  - It is well-defined which parts should check which conditions
  - *Simplifies the program*

51.2. The highly responsible program

Lecture 13 - slide 11

Before we proceed to the role of preconditions and postconditions in relation to responsibility, we will study an example of an object-oriented program with two classes that altogether are over-responsible.

We make our points with yet another version of class `BankAccount`, see Program 51.2, in relation to a client of class `BankAccount`, see Program 51.1. As you will realize below, the illustration of over-responsibility is slightly exaggerated in relation to a real-life program.

The `Main` method in Program 51.1 withdraws and deposits money on the bank account referred by the variable `ba`, which is declared and initialized in line 5. Before withdrawing money in line 8, `Main` checks the soundness of the account (with `AccountOK`), and it checks if there are enough money available. After the withdrawal `Main` checks if the account is still sound. It also deals with the situation where `Main` withdraws an amount of money, which is greater then the balance of the account. Similar observations apply to `Deposit` in line 19.

```
1  public class Client{
2  
3    public static void Main(){
4  
5      BankAccount ba = new BankAccount("Peter");
6  
7      if (ba.AccountOK && ba.EnoughMoney(1000))
8        ba.WithDraw(1000);
9      else
10       WithdrawingProblems("...");
11     if (!ba.AccountOK)
12       MajorProblem("...");
13     if (ba.Balance <= 0)
14       BankAccountOverdrawn(ba);
15 
16     ...
17 
18     if (ba.AccountOK)
19       ba.Deposit(1500);
20     if (!ba.AccountOK)
21       MajorProblem("...");
22 
23   }
24 }
```

Program 51.1 *Excerpt of highly responsible class Client of BankAccount.*

In class `BankAccount` below, the `Withdraw` method in line 9-16 check the soundness of the bank account, and it deals with insufficient funds, before the actual withdrawal takes place in line 15.

The `Deposit` method in line 18-24 cares about the situation where clients deposit very large amounts. In such cases the bank account attempts to check if the money comes from illegal or criminal sources.

```
1  public class BankAccount {
2  
3     private double interestRate;
4     private string owner;
5     private double balance;
6  
7     // ...
8  
9     public void Withdraw (double amount) {
10       if (!AccountOK)
11         ComplainAboutNonValidAccount();
12       else if (!this.EnoughMoney(amount))
13         ComplainAboutMissingMoney();
14       else
15         balance -= amount;
16    }
17 
18    public void Deposit (double amount) {
19       if (amount >= 10000000)
20         CheckIfMoneyHaveBeenStolen();
```

```
      else if (!AccountOK)
        ComplainAboutNonValidAccount();
      alse balance += amount;
   }
}
```

Program 51.2 *Excerpt of highly responsible class BankAccount.*

Seen altogether, the amount of code in Program 51.1 and Program 51.2 is much larger than desired. The checks that happen more than once should be eliminated. In addition, some of the responsibilities should be delegated to third party objects.

51.3. Responsibility division by pre and postconditions

Lecture 13 - slide 12

Preconditions and postconditions can be used to divide the responsibility between classes in an object-oriented program. The idea is to make it the responsibility of particular objects to fulfill the precondition of a method, and to make it the responsibility of other objects to fulfill the postcondition of a method. The rules are as follows:

- Fulfillment of the precondition
  - The responsibility of the caller
  - The responsibility of the *client* in an object-oriented program
- Fulfillment of the postcondition
  - The responsibility of the called operation
  - The responsibility of the *server* in an object-oriented program

Client and *server* are roles of objects relative to the message passing in between them. The client and server roles were discussed in Section 2.1. In some books, the server is called a supplier.

Let us recall the precondition and the postcondition of the square root function `sqrt`, as shown in Program 49.2. A function that calls `sqrt` is responsible to pass a non-negative number to the function. If a negative number is passed, the square root function should do nothing at all to deal with it. If, on the other hand, a non-negative number is passed to `sqrt`, it is the responsibility of `sqrt` to deliver a result which fulfills the postcondition. Thus, the caller of `sqrt` should do nothing at all to check or rectify the result.

Now we know who to blame if an assertion fails:

Blame the caller if a precondition of an operation fails

Blame the called operation if the postcondition of an operation fails

51.4. Contracts

Lecture 13 - slide 13

In everyday life, a contract is an enforceable agreement between two (or more) parties. Often, contracts are regulated by law. In relation to programming in general, we define a contract in the following way:

A *contract* expresses the mutual obligations in between parts of a program that cooperate about the solution of some problem

In object-oriented programming it is natural that the program parts are classes.

The preconditions and the postconditions of the public methods in a class together form a contract between the class and its clients.

It can be a serious matter if a contract is broken. A broken contract is tantamount to an inconsistency between the specification and the program, and it is usually interpreted as an error in the program. The error is usually fatal. A broken contract should raise and throw an exception. Unless the exception is handled, the broken contract will cause the program to stop.

51.5. Everyday Contracts

Lecture 13 - slide 14

Contracts are all around us in our everyday life

When we do serious business in our everyday life, we are very much aware of contracts. When we accept a new job or when we buy a house, the mutual agreement is formulated in a contract.

Below we list some additional everyday contracts:

- Student and University
    - The student enrolls some course
    - The university offers a teacher, a room, supervision and other resources
- Citizen and Tax office
    - The citizen does a tax return
    - The tax office calculates the taxes, and regulates the paid amount of money
- Football player and Football club
    - The player promises to play 50 games per season
    - The football club pays 10.000.000 kroner to the player pr. month
- Citizen and Insurance company
    - The insurance holder pays the insurance and promises to avoid insurance fraud
    - In case of a damage or accident, the insurance company pays compensation

51.6. Contracts: Obligations and Benefits

Lecture 13 - slide 15

Contracts in object-oriented programs, specified by preconditions and postconditions of certain methods, express obligations and benefits.

In Figure 51.1 we personalize the obligations and benefits of a client and server. In the context of Figure 51.1 the server is called a supplier. This terminology, as well as the syntax used in the illustration, come from the object-oriented programming language Eiffel [Meyer97, Meyer92, Switzer93].

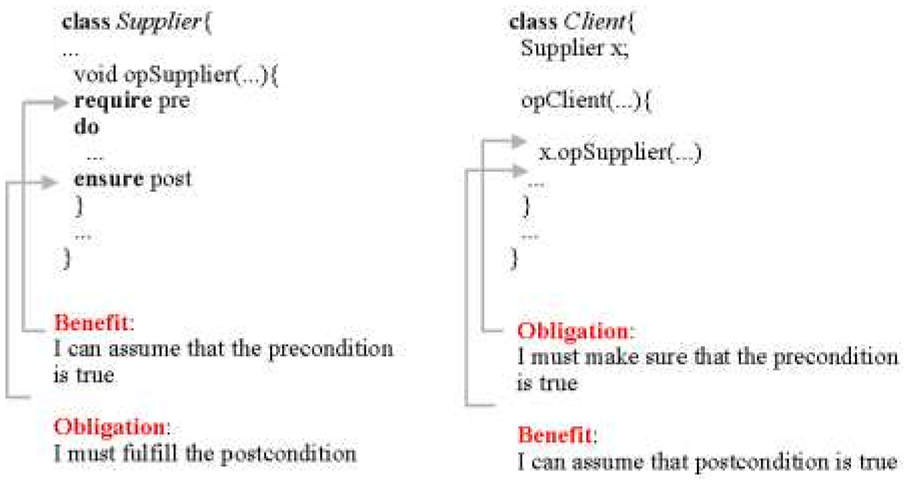

Figure 51.1 *A give-and-take situation involving a client and a server (supplier) class.*

The Client, shown to the right in Figure 51.1 must make an effort to arrange, that everything is prepared for calling `opSupplier` in the class `Supplier`. These efforts can be enjoyed by the supplier, because he can take for granted that required precondition of `opSupplier` is fulfilled.

The roles are shifted with respect to the rest of the game. The supplier must make an effort to ensure that the postcondition of `opSupplier` is fulfilled when the operation terminates. This reflects the fact the operation has done the job, as agreed on in the contract. In return, the client can take for granted that the opposite party (the supplier) delivers an appropriate and correct result.

The obligations and benefits of the contract can be summarized as follows:

Obligation - May involve hard work

Benefit - A delight. No work involved

If you feel that the discussion in this section is too abstract, we will rephrase the essence in the next section relative to the squareroot function.

51.7. Obligations and Benefits in Sqrt

Lecture 13 - slide 16

In Program 49.2 of Section 49.3 we exemplified axiomatic specifications with a squareroot function. Let us, of convenience, rephrase the specification here.

```
sqrt(x: Real) -> Real

  precondition: x >= 0;

  postcondition: abs(result * result - x) <= 0.000001
```

Program 51.3 *An axiomatic specification of the squareroot function.*

The obligations and benefits of `sqrt`, relative to its callers, are summarized in the following table:

-	**Obligation**	**Benefit**
Client	Must pass a non-negative number	Receives the squareroot of the input
Server	Returns a number r for which r * r = x	Take for granted that x is non-negative

Table 51.1 *A tabular presentation of the obligations and benefits of the squareroot function (in a server role) and its callers (in a client role).*

Notice in particular the obligation of the client and the benefit of the server, as emphasized using the **red** color in the table.

51.8. Design by Contract

Lecture 13 - slide 27

As presented in Section 51.4, a contract of a class is the sum of the assertions in the class. Thus, a contract is formed by concrete artifacts in the source program.

As part of the Eiffel efforts [Meyer97, Meyer92, Switzer93], the use and benefits of contracts have been broadened such that contracts affects both design, implementation, and testing. The broad application of contract is known as *Design by Contract* (DBC). Design by Contract is a trademark of the company Eiffel Software, and as such it may be problematic to use the term, at least in commercial contexts.

Design by Contract^{TM} (DBC) represents the idea of designing and specifying programs by means of assertions

The following summarizes the use of contracts in the different phases of the software development process, and beyond.

- **Design:** A pragmatic approach to program specification
- **Documentation:** Adds very important information to interface documentation of the classes
- **Implementation:** Guides and constrains the actual programming
- **Verification:** The program can be checked against the specification every time it is executed
- **Test:**
    - Preconditions limit the testing work
    - The check of postconditions and class invariants do part of the testing work
- **End use:** Trigger exception handling if assertions are violated

The use of contracts for design purposes is central. The contract of a planned class serves as the *specification* of the class. We have discussed program specifications in Section 49.3 of this material.

Interface documentation - as pioneered by JavaDoc - includes signatures of methods and informal explanations found in so-called documentation comments. It is very useful to include both preconditions, postconditions, and class invariants in such documentation.

During program execution - both in the testing phase and in the end use phase - the actual state of the program execution can be compared with the assertions. As such, it is possible to verify the implementation against the specification at program run-time. If an inconsistency is discovered during testing, we have located an error. This is always a pleasure and a success. If an inconsistency is discovered during end use, an exception is thrown. This is clearly less successful. Exceptions have been treated in Chapter 33 - Chapter 36 of this material.

51.9. References

[Switzer93]	Robert Switzer, *Eiffel and Introduction*. Prentice Hall, 1993.
[Meyer92]	Bertrand Meyer, *Eiffel the Language*. Prentice Hall, 1992.
[Meyer97]	Bertrand Meyer, *Object-oriented software construction, second edition*. Prentice Hall, 1997.

52. Class Invariants

In this chapter we will study yet another kind of assertions called class invariants. The class invariant serves as a strengthening of both the preconditions and the postconditions of all operations in the class. As we will see in the first section of this chapter, a good class invariant makes it easier to formulate both preconditions and postconditions of the operations in the class.

52.1. General aspects of contracts

Lecture 13 - slide 18

When we do computations in general, the values of the variables in the running programs are modified throughout the computation. In an object-oriented program the states of the involved objects will vary as the program execution progresses. This variation of the program state is not arbitrary, however. There is usually some rules that control and constrain the variations. Such rules can be formulated as *invariants*. An invariant describes some properties and relationships that remain constant (do not vary) during the execution of a program.

A *class invariant* is an assertion that captures the properties and relationships, which remain stable throughout the life-time of instances of the class.

A *class invariant* expresses properties of an object which are stable in between operations initiated via the public client interface

The following characterizes a class invariant:

- acts as a general strengthening of both the precondition and postcondition
- expresses a "*health criterion*" of the object
- must be fulfilled by the constructor
- must be maintained by the public operations
- must not necessarily be maintained by private and protected operations

The class invariant is an assertion, which should be true at *every stable point in time* during the life of an object. In this context, a stable point in time is just after the completion of the constructor and in between executions of public operations on the class. At a stable point in time, the object is in rest - the object is not in the middle of being updated. The unstable points in time are, for instance, in the middle of the execution of a constructor, or in the middle of the execution of a public operation. In addition, a non-public operation may leave the object in a state, which does not satisfy the class invariant. The reason is that a public operation may need to activate several non-public operations, and it may need to carry out additional state changes (assignments) in order to reach a stable state that satisfies the class invariant. A non-public operation may be responsible for only a fraction of the updating of an object.

You can think of the class invariant as a health criterion, which must be fulfilled by all objects in between operations. As a precondition of every public operation of the class, it can therefore be assumed that the class invariant holds. In addition, it can be assumed as a postcondition of every public operation that the class invariant holds. In this sense, the class invariant serves as a general strengthening of both the precondition and the postcondition of public operations in the class. The *effective precondition* is the formulated precondition in conjunction with the class invariant. Similarly, the *effective postcondition* is the formulated postcondition in conjunction with the class invariant.

A class invariant expresses some constraints that must be true at every stable point in time during the life of an object

Our primary interest in this chapter is class invariants. Invariants are, however, also useful and important in other contexts.

52.2. Everyday invariants

Lecture 13 - slide 19

Before we proceed to a programming example, we will draw the attention to useful everyday invariants.

- **Coffee Machine**
    - *In between operations there is always at least one cup of coffee available*
- **Toilet**
    - *In between "transactions" there is always at least 0.75 meter of toilet paper on the roll*
- **Keys and wallet**
    - *In between using keys and/or wallet*
        - *During daytime: Keys and wallet are in the pocket*
        - *During nighttime: Keys are wallet are located on the bedside table or underneath the pillow*

The coffee machine invariant ensures that nobody will go for coffee in vain. If you happen to fill your jug with the last cup of coffee from the coffee pot, your operation on the coffee machine is not completed until you have brewed a new pot of coffee.

The toilet paper invariant should be broadly appreciated. As a consequence of the invariant, the operation of emptying the toilet paper reel is not completed before you have found and mounted an extra, full reel of paper.

The last everyday invariant is - in my experience - often broken by women and children, because they do not always wear practical cloth with pockets suitable for wallets and keys. As a consequence, these important items tend to be forgotten or misplaced, such that they are not available when needed. If the proposed key and wallet invariant is observed, you either use the key or wallet, or you will be confident where to find them.

Adherence to invariants is the key to order in our daily lives

52.3. An example of a class invariant

Lecture 13 - slide 20

It is now time to study the invariant of the circular list. Recall that we introduced preconditions and postconditions of the circular list in Program 50.1 of Section 50.2.

The class invariant of a circular lists expresses that the list is circular whenever it is non-empty. In Program 52.1 the invariant is formulated at the bottom of the program, in line 43-45. In the same way as the preconditions and postconditions, the class invariant involves subexpressions that are realized by programmed operations (`empty`, `isCircular`, and `size`) of the class.

```
1  class CircularList {
2  
3    // Construct an empty circular list
4    public CircularList()
5      require true;
6      ensure empty();
7  
8    // Return my number of elements
9    public int size()
10     require true;
11     ensure (size = countElements) && noChange;
12 
13   // Insert el as a new first element
14   public void insertFirst(Object el)
15     require !full();
16     ensure !empty() && isFirst(el);
17 
18   // Insert el as a new last element
19   public void insertLast(Object el)
20     require !full();
21     ensure !empty() && isLast(el);
22 
23   // Delete my first element
24   public void deleteFirst()
25     require !empty();
26     ensure (empty() || isFirst(old retrieveSecond));
27 
28   // Delete my last element
29   public void deleteLast()
30     require !empty();
31     ensure (empty() || isLast(old retrieveButLast()));
32 
33   // Return the first element in the list
34   Object retrieveFirst()
35     require !empty();
36     ensure isFirst(result) && noChange;
37 
38   // Return the last element in the list
39   Object retrieveLast()
40     require !empty();
41     ensure isLast(result) && noChange;
42 
43   invariant
44     !empty() implies isCircular()  and
45     empty() implies (size() = 0);
46 }
```

Program 52.1 *Circular list with a class invariant.*

If we compare Program 52.1 with Program 50.1 it is worth noticing that the preconditions and postconditions become simpler and shorter, because they implicitly assumes that the class invariant is true. Thus, relative to (a slightly idealised version of) Program 50.1, the invariant is factored out of all preconditions and postconditions.

53. Inheritance is Subcontracting

In this chapter we will review inheritance - including specialization - in the light of contracts. Specialization was discussed in Chapter 25 and inheritance was discussed in Chapter 27. The concept of contracts was introduced in Chapter 51.

Stated briefly, we understand a subclass as a *subcontractor* of its superclass. Being a subcontractor, it will not be possible to carry out arbitrary redefinitions of operations in a subclass, relative to the overridden operations in the superclass.

53.1. Inheritance and Contracts

Lecture 13 - slide 22

The following question is of central importance to the discussion in this chapter.

How do the assertions in a subclass relate to the similar assertions in the superclass?

Figure 53.1 illustrates a class B which inherits from class A. Both class A and B have invariants. In addition, operations in class A that are redefined in class B have preconditions as well as postconditions.

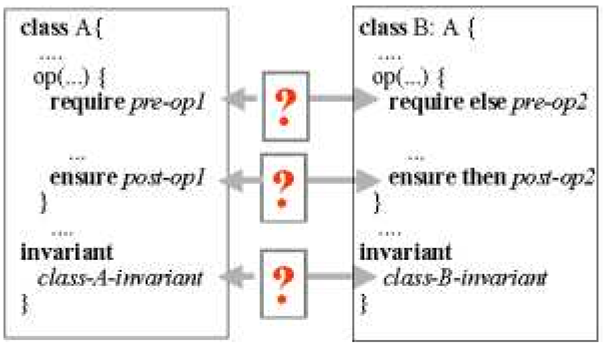

Figure 53.1 *The relationship between inheritance and contracts*

The question from above can now to refined as follows:

- How is the invariant in class B related to the invariant of class A?
- How is the precondition of the operation `op` in class B related to the precondition of the overridden operation `op` from class A?
- How is the postcondition of the operation `op` in class B related to the postcondition of the overridden operation `op` from class A?

Each of the three questions are symbolized with a red question mark in Figure 53.1.

53.2. Subcontracting

Lecture 13 - slide 23

Due to polymorphism, an instance of a subclass can act as a *stand in* for - or *subcontractor* of - an instance of the superclass. Consequently, the contract of the subclass must comply with the contract of the superclass. The contract of a subclass must therefore be a *subcontract* of the superclass' contract. This is closely related to the principle of substitution, which we discussed in Section 25.7.

The notion of subcontracting is realized by enforcing particular requirements to preconditions, postconditions, and class invariants across class hierarchies. In order to understand inheritance as subcontracting, the following rules must apply for assertions in a subclass:

- The precondition must not be stronger than the precondition in the superclass
- The postcondition must not be weaker than the postcondition in the superclass
- The class invariant must not be weaker than the invariant in the superclass

As discussed in Section 50.1, a precondition of an operation states the prerequisites for calling the operation. If the precondition is evaluated to the value *true*, the operation can be called. It is the responsibility of the caller (the client) to fulfill the precondition. The postcondition of the operation states the meaning of the operation, in terms of requirements to the returned value and/or requirements to the effect of the operation. It is the responsibility of the operation itself (the server) to fulfill the postcondition. The postcondition must be *true* if the precondition is satisfied and if the operation terminates normally (without throwing an exception).

If we assume that the precondition of a redefined operation in a subclass is stronger than the precondition of the original operation in the superclass, then the subclass cannot be used as a subcontractor of the superclass. Consequently, the preconditions of redefined operations in subclasses must be equal to or weaker than the preconditions of corresponding operations in superclasses.

In case the postcondition of a redefined operation in a subclass is weaker than the postcondition of the operation in the superclass, the redefined operation does not solve the problem as promised by the contract in the superclass. Therefore, the postconditions of redefined operations must be equal to or stronger than the postconditions of corresponding operations in superclasses.

The superclass has promised to solve some problem via the virtual operations. Redefined and overridden operations in subclasses are obliged to solve the problem under the same, or possible weaker conditions. This causes the weakening of preconditions. The job done by the redefined and overridden operations must be as least as good as promised in the superclass. This causes the strengthening of postconditions.

The invariant of the superclass expresses requirements to instance variables in the superclass, at stable points in time. These instance variables are also present in subclasses, and the requirements to these persist in subclasses. Consequently, class invariants cannot be be weakened in subclasses.

Relative to Figure 53.1 the formulated precondition `pre-op2` in `B.op` serves as a weakening of `pre-op1` of `A.op`. The effective precondition of `B.op` is `pre-op1` **or** `pre-op2`. Similarly, the effective postcondition of `B.op2` is `post-op1` **and** `post-op2`. The use of the Eiffel keywords **require else** and **ensure then** signals this understanding.

Operations in subclasses cannot arbitrarily redefine/override operations in superclasses

In our discussion of redefinition of methods in Section 28.9 we came up with some technical and syntactical requirements to redefinitions. The contributions outlined above in terms of subcontracting constrain the meaning (the semantics) of redefined operations in subclasses in relation to the original operations in superclasses. This is very satisfactory!

53.3. Class invariants in the triangle class hierarchy

Lecture 13 - slide 24

We studied the specialization hierarchy of polygons in Section 25.5. In Figure 53.1 below we revisit the five triangle classes. It is our interest to understand how the class invariants are strengthened in subclasses of the most general triangle class.

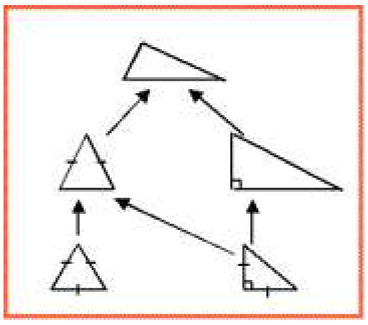

Figure 53.2 *The hierarchy of triangle classes. The root class represents the most general triangle. The son to the left represents an isosceles triangle (where to sides are of same lengths). The son to the right represents a right triangle, where one of the angles is 90 degrees. The triangle at the bottom left is an equilateral trianlge (where all three sides are of equal lengths). The triangle at the bottom right is both an isosceles triangle and a right triangle.*

The invariants of the five types of triangles can be described as follows:

- **Most general triangle:**
  3 angles, 3 edges
  Sum of angles: 180 degrees
- **Isosceles triangle**
  Invariant of general triangle
  2 edges of equal length
- **Equilateral triangle:**
  Invariant of isosceles triangle
  3 edges of equal length
- **Right triangle:**
  Invariant of general triangle
  Pythagoras
- **Isosceles right triangle:**
  Invariant of isosceles triangle
  Invariant of right triangle

Notice that the *italic contributions* above describe the strengthenings relative to the invariant of the superclass.

53.4. Assertions in Abstract classes

Lecture 13 - slide 25

Abstract classes where discussed in Section 30.1. An abstract method in an abstract class defines the name and parameters of the method - and nothing more. The intended meaning of the method is an informal matter. In Chapter 30 we did not encounter any means to define or constrain the actual result or effect of abstract methods. In this section we will see how the meaning of an abstract method can be specified.

In Program 30.1 we studied an abstract class `Stack`. Below, in Program 53.1 we show a version of the abstract stack with contractual elements - preconditions and postconditions. Possible future non-abstract subclasses of `Stack` will be subcontractors. It means that such subclasses will have to fulfill the contract of the abstract stack, in the way we have discussed in Section 53.2.

```
using System;

public abstract class Stack{

  abstract public void Push(Object el);
    require !full;
    ensure  !empty  &&  top() = el  &&  size() = old size() + 1 &&
            "all elements below el are unaffected";

  abstract public void Pop();
    require !empty();
    ensure  !full()  &&  size() = old size() - 1 &&
            "all elements remaining are unaffected";

  abstract public Object Top
    require !empty();
    ensure  nochange  &&  Top = "the most recently pushed element";    {
    get; }


  abstract public bool Full
    require true;
    ensure nochange  &&  Full = (size() = capacity);   {
    get; }


  abstract public bool Empty
    require true;
    ensure nochange  &&  Empty = (size() = 0);   {
    get;}

  abstract public int Size
    require true;
    ensure nochange  &&  Size = "number of elements on stack";   {
    get;}

  public void ToggleTop()
    require size() >= 2;   {
    if (Size >= 2){
      Object topEl1 = Top;  Pop();
      Object topEl2 = Top;  Pop();
      Push(topEl1); Push(topEl2);
    }
    ensure size() = old size() &&
    "top and element below top have been exchanged"  &&
    "all other elements are unaffected";
  }

  public override String ToString(){
    return("Stack");
  }
}
```

Program 53.1 *An abstract class with preconditions and postconditions.*

As we have seen before, **require** clauses are preconditions and **ensure** clauses are postconditions. Notice the use of **old** and **nochange**, which we introduced in Section 50.2. The *"italic strings"* represent informal preconditions. Alternatively, and more rigidly, we may consider to implement these parts of the assertions as private boolean functions. Notice, however, that it would be quite demanding to do so, at least compared with the remaining implementation efforts involved.