Poul Klausen

C# 1

Basic syntax and semantics

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 1: BASIC SYNTAX AND SEMANTICS

SOFTWARE DEVELOPMENT

C# 1: Basic Syntax and Semantics: Software Development
1st edition

ISBN 978-87-403-3346-6

CONTENTS

	Foreword	**6**
1	**Programming**	**8**
2	**Hello World**	**10**
2.1	Visual Studio	10
2.2	The Kings	15
3	**Commands and console programs**	**17**
3.1	Commands	17
3.2	PrintAddress	19
3.3	Console programs	21
4	**Variables and data types**	**24**
4.1	Operators	31
4.2	Literals	38
4.3	Formatting output	42
4.4	Objects	46

4.5	TheAddress	50
4.6	Inheritance	55
4.7	Namespaces	60
4.8	Example: Cubes	65
4.9	Arrays	68
4.10	Example: CupProgram	73
4.11	Multidimensional arrays	76
5	**Program control**	**80**
5.1	The if statement	80
5.2	do and while statements	85
5.3	The for statement	88
5.4	The switch statement	95
5.5	return statement	98
5.6	break, continue and goto	98
6	**List**	**104**
7	**Comparison and sorting**	**107**
8	**Files**	**115**
8.1	Text files	115
8.2	Serialization of objects	119
9	**More about classes**	**124**
9.1	The program stack	124
9.2	Class examples	130
9.3	More classes	135
9.4	Methods	151
9.5	Objects	155
9.6	Visibility	156
9.7	Static members	157
10	**Final example: Lotto**	**161**
10.1	Design	163
10.2	Programming and test	167

FOREWORD

This book is the first in a series of books on software development. The programming language is C#, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. The subject of the current book is an introduction to the programming language C# with an emphasis on basic language syntax and semantics, but it is also a book about what programming in general is and how to practically write and test simple programs. The book requires no knowledge about programming or the language C#, and the goal is to show how to get started writing computer programs. After reading the book and worked through the book's exercises and problems, the reader should be able to write simple console applications in the language C#.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in C#. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it, and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples, exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code for the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with a larger sample program, which focus primarily is on the process and an explanation of how the program is written. On the other hand appears the code only to a limited extent, if at all, and the reader should instead study the finished program code, perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

All sample programs are developed and tested on a Windows machine, and the development tool is Visual Studio. Therefore, you must have Visual Studio installed and Microsoft provides a free version that is fully adequate. Visual Studio is an integrated development environment that provides all the tools needed to develop programs. I do not want to deal with the use of Visual Studio, which is quite straightforward, and it is a program that you quickly become familiar with.

Finally a little about what the books are not. It is not "a how to write" or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 PROGRAMMING

A computer program is a series of commands executed in a certain order, and together they solve a specific task. A program is written as a text document that contains all the necessary commands. This document is called the program code or the source code. The individual commands must be written in a very precise way for the computer to understand them, and it is here programming languages comes into the picture. A programming language defines precise rules for how the commands should be written. There are many programming languages, and although they are different, each with their advantages and disadvantages, the similarities outweighs the differences, and once you have learned one language, it is easy to learn the next. In the following are used throughout the C# programming language, which is a widely used language on Windows platforms. How the individual commands and orders exactly must be written is called the language's syntax. What the individual commands are doing or performing is called the language's semantics.

As mentioned above, a program is written as a text document (in practice several or many), that is simply a document of commands. Commands are also called statements. These commands or statements being only text, the machine can not immediately perform the commands, but they must be translated into an internal format that the computer understands. This process is called translation or *compilation* and is performed by a program that can convert statements written in a particular programming language to the computer's internal commands. The program is usually called a *compiler*. During the translation the program is checked for errors, and if there are errors, you get an error message, and the errors must then be corrected before the program is translated again. Not all errors are found during translation, but only syntax errors, which covers the issue where a statement is not written in accordance with the programming language's rules. A translated program can easily contain other errors, for example a miscalculation.

To write a program you must of course learn the programming language that is selected, but also you must learn how the solution of a task can be formulated by statements in the language. It is the latter that is the most difficult, and there is rarely a clear solution. A solution of a problem by means of a program is also called an *algorithm*. Programming is largely a matter of writing algorithms, something that I will return to several times.

When you have to write software, you need a tool that can be used to enter the program code, and in principle you could do that with a simple input program (a text editor as *Notepad*) and then the compiler, but in practice you will always use a specific development tool, as it makes the job much easier. In the following I will everywhere use *Visual Studio*, which is Microsoft's tool for software development and is a tool used to write programs in a variety of programming languages and also C#. It is an integrated software package, which includes all the tools necessary for the development of a number of different types of programs.

C# is an object oriented programming language. The fundamental architectural element in a program is a *class*, and from the programmer's point of view a C# program consists of a family of classes that collectively defines all the application's features and functionality. Writing a program is thus to define, design and write the code for the program's classes. Nothing in C# exists outside of a class. A program will also always apply other classes that are not written by the programmer, but classes coming from the .NET API, and thus is available for the programmer as finished components. One of the program's classes have a special role as the program's *entry point* and the place where the program starts and this class should be written with a special syntax, but it is almost the only formal requirements for the architecture of a C# program.

In this series of books the platform is Windows, and programs are written in the programming language C#, but a compiled C# program can not immediately be performed on the Windows machine. It needs a runtime system, which is a program installed on the Windows machine (and will automatically be) that can executes compiled C# programs. This program is called the .NET runtime, and is a program which can interpret and execute translated C# programs. The .NET runtime system can also execute programs written in other .NET languages and in this way you can write programs where different parts of the program are written in different programming languages.

It is the programmer's job to write the program as text in the selected programming language, which here is C#. After that, the program must be translated to the .NET platform, and if there are errors because the programmer did not write the program correctly, the errors must be corrected and the program translated again. For this work you use a special development tool, which is called an IDE (for integrated development environment), and the purpose is to gather a number of tools that make writing a program easier. In practice, you do not write programs without using such an IDE, and in the following it will everywhere be Visual Studio, which is Microsoft's integrated development tool.

2 HELLO WORLD

The subject of this chapter is to show how to write and run a C# program using Visual Studio and the aim is solely to get started. There are several kinds of programs, or you can say that the programs can be categorized in several ways, but in the first books I will look at three types of programs:

- *commands*, which is a program that is typically performed from a command window when you enter the program name that may optionally be followed by one or more arguments
- *console applications*, who also is performed at a command prompt, but here the program runs in a dialogue with the user, where the user must enter values when the program is executed
- *GUI-programs*, where the program opens one or more windows with components as buttons and input fields used by the user to interact with the program

This division is only for practical reasons, as I will sometimes characterize the program examples in relation to this, but common to the three types of programs is that they are standalone applications that run on a single machine and without using resources on other machines.

I'll start with the classic *Hello World* program, a program that prints a text on the screen. It is an example of a command, but it is also an example of a program that has absolutely no practical interest. Although it is a simple program, it will nevertheless treat a number of basic principles that apply to all C# programs.

2.1 VISUAL STUDIO

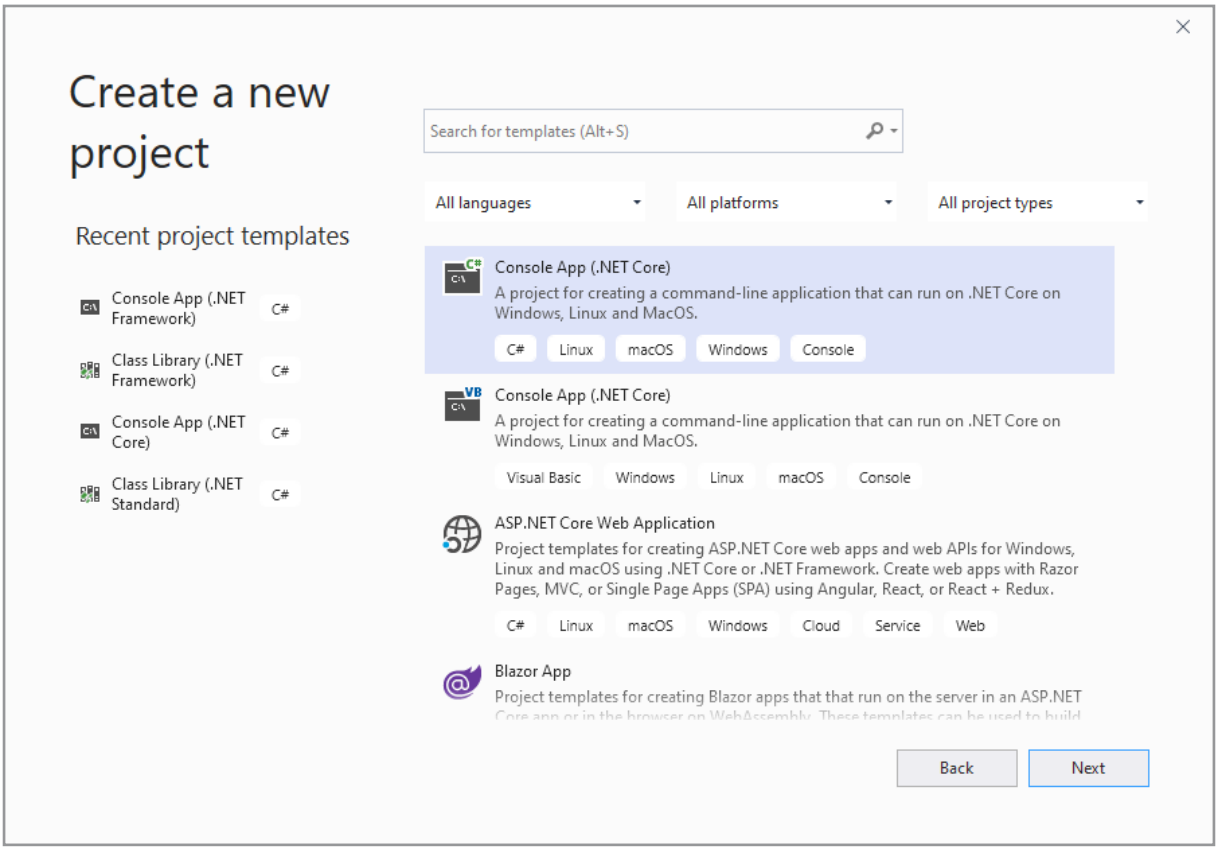
As mentioned a C# program is written as text files, which will then be translated. When the files are translated without errors the program can be executed by the .NET runtime. To write the first C# program, open Visual Studio and create a new project. In the menu, choose

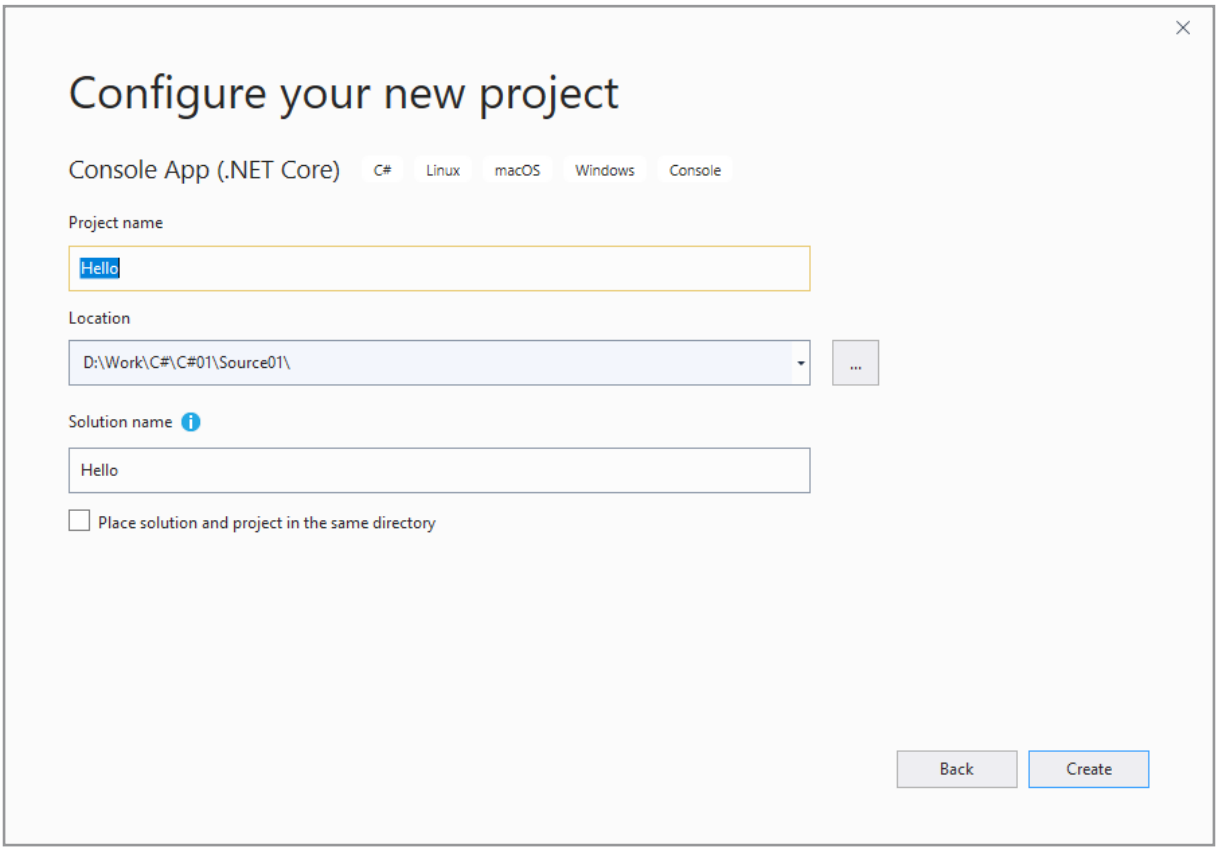
File | *New Project* (see below)

A Visual Studio project creates all the necessary files required to develop and test the program and eliminates a variety of configurations that are otherwise necessary. Using Visual Studio you can build and run the program just by clicking the menu. To create the project, you must select the project type, and here you must select *Console App (.NET Core)* (see below).

In the next window you must give the project a name and select where in the file system to create the project. I have called the project *Hello*. When you then click *Create* Visual Studio creates the project and a file with the following content:

```
using System;

namespace Hello
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```





It is the code skeleton for a program written in C#, and in fact it is a fully finished program that can be translated and run on the machine. In the menu there is a green arrow followed by the word *Hello*. If you click on the item, the program is executed and Visual Studio opens the following window:

Note, that the program writes the text *Hello World!* (the rest of the text you can ignore as it has nothing with the program to do).

If you look at the program code above you should note that C# is case-sensitive, so that everywhere you have to distinguish between uppercase letters and lowercase letters.

Every C# program consists of at least one class, here called *Program* (the name chosen by Visual Studio). A class consists of variables and methods. In this case, the class has only one method called *Main()*, which is the method called when the program starts. A method consists of statements that can be perceived as commands that are performed on the machine. That a method is called means that its statements are executed. Note that the

method *Main()* must be preceded by the word *static*. The explanation of that comes later. In this case, *Main()* has only a single statement, writing a text on the screen. *WriteLine()* is actually a method in a class *Console*. When the program runs, there is nothing else than the *WriteLine* statement in *Main()* which print a text on the screen.

Note that in C#, every statement ends with a semicolon, above there is a semicolon after the *WriteLine* statement. It tells the compiler where the statement ends.

In C# classes are grouped in so-called namespaces. *System* is a namespace that contains many classes including the class *Console*. A class's full name consists of the namespace where the class is grouped, and the class name, for example *System.Console*. In a program *using* defines a namespace, and classes in this namespace can be referenced by the class name alone. Thus, one can write

```
Console.WriteLine("Hello World");
```

instead of the full name

```
System.Console.WriteLine("Hello World");
```

Visual Studio will automatically place the program in its own namespace, here called *Hello*. If you wrote the program using a plain text editor, it is not necessary to include this namespace. Actually the program can be written simpler than the above. The following version of the program is written in Notepad and saved as a file named *Hello.cs*:

```
class Program
{
  static void Main()
  {
    System.Console.WriteLine("Hello World");
  }
}
```

If you then open a .NET prompt, the program can be translated with the command

```
csc Hello.cs
```

and the result is a file *Hello.exe*, which is an executable file that can be tested. All program examples in this book is written in Visual Studio, since the gain from bigger programs are considerable, but in fact *csc* (the C# compiler) is the only required tool for developing .NET applications.

When you creates the program using Visual Studio it creates a project containing many directories and files. My project has the name:

```
D:\C#\C#01\Source01\Hello
```

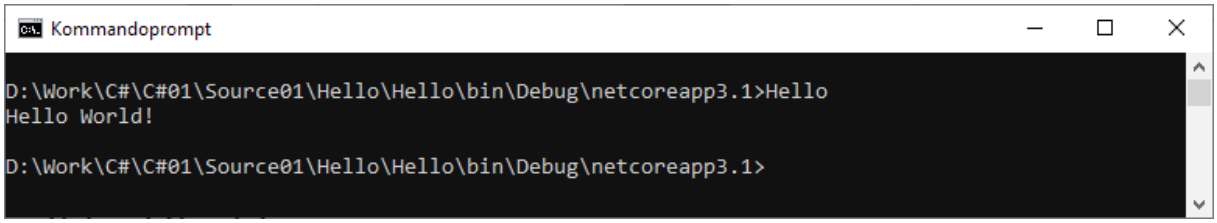
If you look in the directory you will find the file:

```
D:\C#\C#01\Source01\Hello\Hello\Program.cs
```

It is the file with the code shown above and is called the source file. If you look in the directory

```
D:\C#\C#01\Source01\Hello\Hello\bin\Debug\netcoreapp3.1
```

you will find 6 files and here the files *Hello.exe* and *Hello.dll*. It are the files with the code for the compiled program, while the other files are configuration files. If you opens a command prompt and change current directory to the above directory you can execute the program:



2.2 THE KINGS

The example is a program called *Kings*, in principle written in the same way as *HelloWorld*, but the program prints the following text:

```
##################################
# Gorm den gamle  #      # 958    #
# Harald Blåtand  #      # 987    #
# Svend Tveskæg   #      # 1014   #
# Harald d. 2.    # 1014 # 1018   #
# Knud den Store  # 1018 # 1035   #
##################################
```

The final code is shown below:

```
using System;

namespace Kings
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("##################################");
      Console.WriteLine("# Gorm den gamle  #      #  958 #");
      Console.WriteLine("# Harald Blåtand  #      #  987 #");
      Console.WriteLine("# Svend Tveskæg   #      # 1014 #");
      Console.WriteLine("# Harald d. 2.    # 1014 # 1018 #");
      Console.WriteLine("# Knud den Store  # 1018 # 1035 #");
      Console.WriteLine("##################################");
    }
  }
}
```

and if you run the program, you get the following result:

The only difference compared to *HelloWorld* is that this time there are several *Console.WriteLine()* statements.

Exercise 1: The Digits program

Write a program, as you can call *Digits*, which on the screen prints the following table:

```
*************
* 1 * One   *
* 2 * Two   *
* 3 * Three *
* 4 * Four  *
* 5 * Five  *
* 6 * Six   *
* 7 * Seven *
* 8 * Eight *
* 9 * Nine  *
*************
```

3 COMMANDS AND CONSOLE PROGRAMS

In the previous chapter I divided the programs into three categories, and in this chapter I will look at commands and console programs. In principle there is no big difference, and the division go alone on how the user is transferring data to the program. Both *HelloWorld* and the example *Kings* from the previous chapter are examples of commands.

3.1 COMMANDS

Every C# program must have a *Main()* method that has the following signature:

```
static void Main(string[] args)
```

For the moment you should ignore the meaning of the words *static* and *void* and just accept that they should be there, but the *Main()* method is the place where the program starts. After the method's name is a parameter in parentheses, indicating arguments from the command line that can be transferred to the program. Consider the following program:

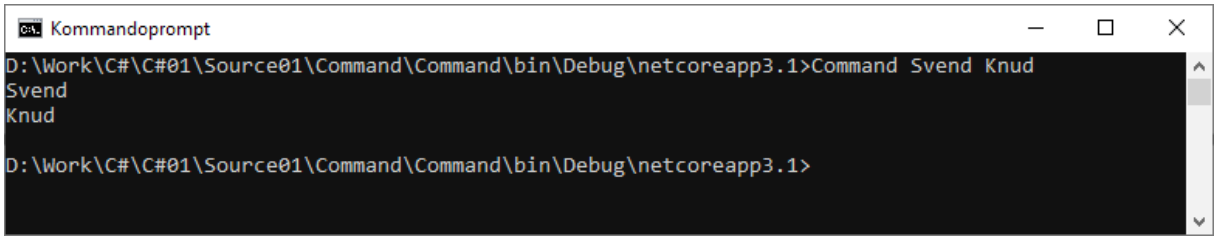
```
using System;

namespace Command
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(args[0]);
      Console.WriteLine(args[1]);
    }
  }
}
```

args is an array, as I explains later, but the arguments that are transferred on the command line, are in the program referred to as

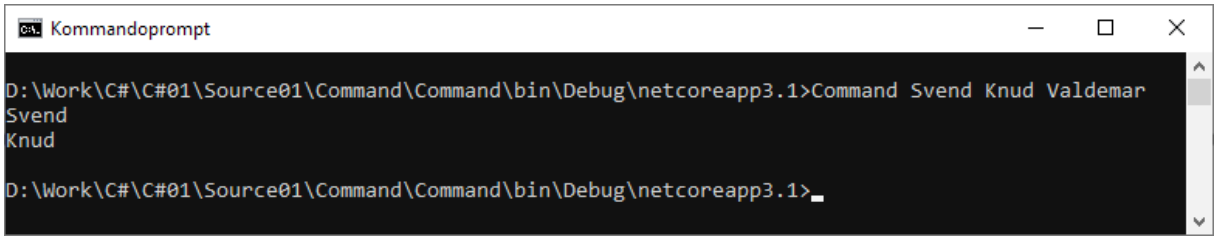
args[0]
args[1]
args[2]

and so on.

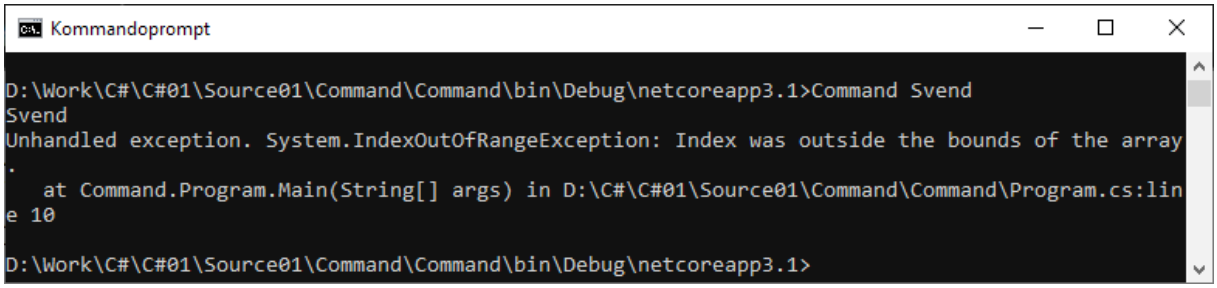
An argument is a text string, and arguments are separated by spaces. Below are examples of running the program with different arguments. In the first case there are two arguments, called respectively *Svend* and *Knud*, and the program prints the two arguments on the screen:



If you enter three arguments, the result is the same:



This time there are three arguments, but only the first two are used in the program. If, however, the application is performed without having two arguments, you get an error:



The reason is that the second argument does not exist, and therefore fails the statement:

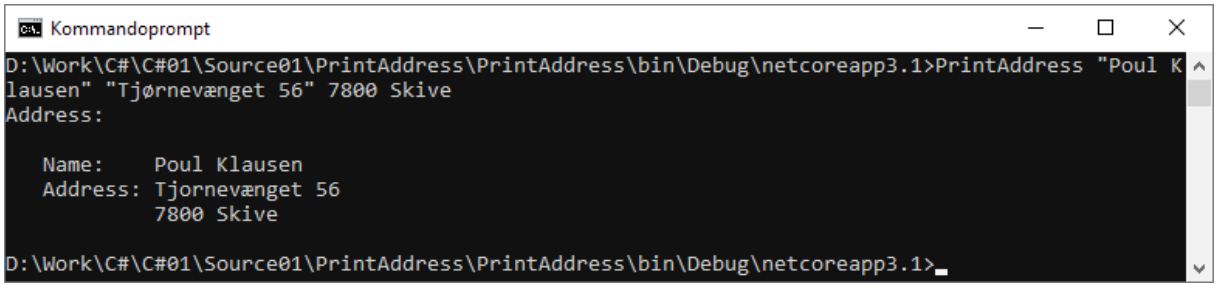
```
Console.WriteLine(args[1]);
```

3.2 PRINTADDRESS

The following program will print the name and address of a person, but so that the values to be printed are transferred on the command line. The program is therefore a command. The program is called *PrintAddress*, and there must be transferred four arguments on the command line. Arguments on the command line are separated by spaces, and if an argument contains spaces, it is necessary to put the argument in quotes. Then the runtime system will perceive it as one argument. An example of running the program could, for example be the following command:

```
PrintAddress "Poul Klausen" "Tjørnevænget 56" 7800 Skive
```

and the program prints the following message om the screen:



To write the program I have in Visual Studio created a project called *PrintAddress*. The program code can then be written as follows:

```
/*
 * Program that on the screen prints a person's name and address.
 * The values are passed as arguments on the command line.
 */
using System;

namespace PrintAddress
{
  /*
   * Enter a person's name and address on the form
   * name address zipcode town
   * that is four fields separated by spaces.
   */
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Address:");
      Console.WriteLine();
      Console.WriteLine(" Name:       " + args[0]);
      Console.WriteLine(" Address:    " + args[1]);
      Console.WriteLine("             " + args[2] + " " + args[3]);
    }
  }
}
```

The statements in the *Main()* method is all *Console.WriteLine()* statements that prints a line on the screen. The first only prints a text, while the second prints a blank line. The third prints a text followed by the value of *args[0]*, which is the person's name. Note particularly the *plus* operator, which means *string concatenation*, where the text is added after the other. The last two *Console.WriteLine()* statements works in principle in the same way.

You should note the comments which is text that explains how the program is written and works. All text between /* and */ are comments, and comments have no effect on the translated program and are ignored by the compiler. Comments are only for those who need to read and understand the program. There are other comments which I will explain later and it may be helpful to provide the code with comments as shown in this example.

A program like the above are not robust, as it will fail, if the user does not transfer the right number of arguments. You should also note that the program does not test the arguments (which incidentally is also not so easy), but simply prints the arguments as they are.

Problem 1: Print a recall

You should write a program for a library that can print a recall of a book. When the program is executed, you must on the command line transfer five arguments:

- borrower's name
- borrower's address
- borrower's zip code and town
- ISBN of the book
- the books title

An example of an execution of the program might be:

```
                              #############
                              #   Recall  #
                              #############

To:

    Poul Klausen
    Tjørnevænget 56
    7800 Skive

When we can see that the loan period for the
following book is the end, we must ask you as quick
as possible return the book to the library.

Title:

    978-0132126953
    Computer Networks

Yours sincerely
The Research Library
```

3.3 CONSOLE PROGRAMS

Compared to a command a console program (in this books) is a program that performs a dialogue with the user, where the user must enter data.

Above I have shown a program that prints the name and address of a person when the values are passed as arguments on the command line. Below is the same program, but this time the user must enter the values while running the program. There is thus a dialogue with the user.

```
using System;

namespace InputAddress
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.Write("Enter name: ");
      String name = Console.ReadLine();
      Console.Write("Enter address: ");
      String addr = Console.ReadLine();
      Console.Write("Enter zipcode: ");
      String post = Console.ReadLine();
      Console.Write("Enter town: ");
      String town = Console.ReadLine();
      Console.WriteLine();
      Console.WriteLine("Address:");
      Console.WriteLine();
      Console.WriteLine(" Name:       " + name);
      Console.WriteLine(" Address:    " + addr);
      Console.WriteLine("             " + post + " " + town);
    }
  }
}
```

As you can see, the code has been substantially larger, and there is also new things that has to be explained.

To enter text, you must have an object that represents the computer and provides a service available for entering text. *Console* is such an object and has a method *ReadLine()* which can be used to enter a text line. The first statement in the program prints a text on the screen, telling the user to enter the name. It happens with the statement

```
String name = Console.ReadLine();
```

The text that the user enters, must be stored somewhere, and for that purpose a *variable* is used. Variables are considered in the next chapter, but a variable is a place where you can store a value, and an example is the statement above.

Here, the text that the user enters is stored in the variable *name*. The next statements are identical in principle and is used for entering the other values. The last statements are used to print the result and is similar in principle to the previous version of the program, but the arguments *args[0]*, *args[1]*, ... are replaced with the variables. If you run the program (from Visual Studio), the result could be:

Today it is rarely, if ever you develop console applications, and when there is a need for a program with a user dialogue (and it is of course often), you write a Windows or GUI program. Console programs may, however, for testing and learning be useful, and therefore it is excellent to know how to write a simple console program.

Problem 2: Another recall

You must solve the same task as in problem 1, but instead of transferring values as arguments on the command line, you must enter information about

- borrower's name
- borrower's address
- borrower's zip code and town
- ISBN of the book
- the books title

when the program is running in a dialogue with the user. The program should print the same recall as in problem 1.

4 VARIABLES AND DATA TYPES

Programs has to deal with data, and for that they need a way where to save or store data. To that purpose programs uses variables that are objects, where the program may store a value. You must note that I have already used variables associated with data entry. A variable is characterized by

- a name
- a type
- operators

Variables must have a name, so you can refer to them in the program. C# is, similar to other modern programming languages, relatively flexible in terms of naming variables, but the following shall (should) be met:

- a variable name should always start with a lowercase letter
- then may follow any number of characters consisting of letters, digits and underscore
- a variable name must not contain spaces
- the name of a variable must not be a reserved word, that is a word that has specific meaning in C#

In addition, a variable name should tell something about what it is used for. Use whole words instead of cryptic abbreviations. It will make the code easier to read and understand. If you have long names consisting of several words, you can increase readability by letting a word (except the first) start with a capital letter such as *customerAddress*. Alternatively, there is some who writes *customer_address*, but generally avoid very long names.

If you do not break these rules, you have never problems with names of variables, but some other characters are actually allowed.

Variables has a type that indicates which values can be stored in them, and how much room a variable is using in the machine's memory. The type determines simultaneously the operations that can be performed on a variable, that is what you can do with it.

Variables must be created or declared before they can be used. This is done by a statement of the form:

type name = value;

That is, you must write the type first, then the variable name and finally assign it a value, for example

```
int number = 23;
```

Here is declared a variable called *number*, that has the type *int* and the value 23. Variables should always be initialized, else you can get a translation error.

When variables must be declared, it is because the compiler must allocate room in the machine's memory, and when the name appears somewhere in the code, the compiler must know what the name means, to check if the variable is used in the proper context. If not, the compiler will come up with an error message. The program can only be used when it is compiled without errors.

C# has the following built-in primitive or simple data types:

Type	Description	Value notations
bool	Boolean	true, false
char	16 bit unicode character	'A', '\x0041', '\u0041'
sbyte	8 bit signed integer	
byte	8 bit unsigned integer	
short	16 bit signed integer	
ushort	16 bit unsigned integer	
int	32 bit signed integer	
uint	32 bit unsigned integer	Suffix: U
long	64 bit signed integer	Suffix: L/l
ulong	64 bit unsigned integer	Suffix: U/u eller L/l
float	32 bit floating-point number	Suffix: F/f
double	64 bit floating-point number	Suffix: D/d
decimal	96 bit decimal number	Suffix: M/m
string	Charater string (text)	"C:\\test.txt", @"C:\test.txt"

The first column tells the type, the second how much a variable of that type fills in the machine memory, and what values it may contain. The last column shows how to declare values of that type. A variable can then contains the following values:

- *sbyte*, which is a data type for an integer. A variable occupies 8 bits, and may contain values from -128 to 127 (both inclusive).
- *byte*, which is also an integer type, but can not contain negative values. A variable occupies 8 bits, and may contain values from 0 to 255 (both inclusive).
- *short*, which is a data type for integers. A variable takes up 16 bits and may contain values from -32768 to 32767 (both inclusive).
- *ushort*, which is a short that only can contain none negative values. A variable takes up 16 bits and may contain values from 0 to 65535 (both inclusive).
- *int*, which is a data type for an integer. A variable occupies 32 bits and can contain values from -2147483648 to 2147483647 (both inclusive). It is the default type for an integer.
- *uint*, which is an *int* for none negative integers. A variable occupies 32 bits and can contain values from 0 to 4294967295 (both inclusive). It is the default type for an integer.
- *long*, which is a data type for an integer. A variable occupies 64 bits and can contain values from -9223372036854775808 to 9223372036854775807 (both inclusive).
- *ulong*, which is a data type for an unsigned integer. A variable occupies 64 bits and can contain values from 0 to 18446744073709551615 (both inclusive).
- *float*, which is a data type for floating point numbers and can thus be used for decimal numbers. A variable occupies 32 bits and can represent decimal numbers with 7-8 significant digits. It is important to note that the value is always a rounded result.
- *double*, which is a data type for floating point numbers and can thus be used for decimal numbers. A variable occupies 64 bits and can represent decimal numbers with 14 significant digits. It is important to note that the value is always a rounded result. This type is the default type for a floating point.
- *bool*, which is a data type with only two values: *false* or *true*. The type is important to be able to write conditions and is used especially for program control.
- *char*, used for characters, and a variable of the type *char* takes up 16 bits. Values are numeric codes with values from 0 to 65535 (both inclusive), and each character is represented by a numeric codes. For example has a large A the code 65.

The last type *string* is slightly different than the others and the type is called a reference type, which is explained later. A value of a *string* can start with a @ character, that means that escape characters are not interpreted. Escape characters are characters in a string that has a special meaning, and they always start with \ followed by a character. For example means \n line break.

The smallest unit you can use on a digital computer is a *bit*, and the whole computer's memory consists of a number of items that can store one bit. One bit can represent one of two values, commonly referred to as 0 and 1, and the content of the computer's memory is always a large number of 0's and 1's. In practice, you can not directly refer to the individual bits, but they are organized in groups of 8 bits, and such a group is called a *byte*. A byte is a pattern consisting of 8 bits, for example

```
00111001
```

What that means depends on how a program uses the pattern. Since each of the 8 places in a byte, may have two values, a byte therefore can represent $2^8=256$ different values. As is clear from the above the simple data types are different in how many bits they uses to a variable of that type. As an example uses an *int* 32 bits (or 4 bytes), and this means that one can represent $2^{32}=4294967296$ different integers that is interpreted as the numbers starting from -2147483648 to and including 2147483647.

A *string* is a type which can contain any text string. As an example you can write

```
private String name = "Knud den Hellige";
```

The statement creates a variable of the type *String* (or *string* which means the same). You should note that you specify the value in quotes. The type *string* is not a primitive type, but it is instead a *class*, but for now you can ignore it, and variables of the type *string* is used in principle in the same way as other variables.

Primitive variables are assigned a default value by the compiler. The type *bool* have the default value *false*, a *char* has a space as the default value, while the other primitive types have the default value 0. A *string* (which is not a primitive type) has no value, which is defined as *null*.

As an example, the following statements creates three variables, all of the type *int*:

```
int number1 = 17;
int number2 = 23;
int sum = number1 + number2;
Console.WriteLine("The sum of " + number1 +
" and " + number2 + " is " + sum);
```

The first two variables are initialized with numbers, while the last is initialized to the sum of the first two. The last statement prints the values of the three variables.

Consider as an example the following program, which is a console program where the user has to enter three numbers which must be integers, and the program prints the sum of the three numbers:

```
using System;

namespace Sum3
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        Console.Write("Enter a number: ");
        int a = int.Parse(Console.ReadLine());
        Console.Write("Enter a number: ");
        int b = int.Parse(Console.ReadLine());
        Console.Write("Enter a number: ");
        int c = int.Parse(Console.ReadLine());
        Console.WriteLine(a + " + " + b + " +
        " + c + " = " + (a + b + c));
      }
      catch (Exception ex)
      {
        Console.WriteLine("Ulovlige argumenter");
      }
    }
  }
}
```

If you execute the program and enter the numbers 13, 17 and 19, the result will be:

```
13 + 17 + 19 = 49
```

Values which you enter on the console, has always the type *String*, and they can not be directly used in a calculation. It is first necessary to convert the values (which is text) to a numeric value. It can, for example be done in the following way:

```
int a = int.Parse(Console.ReadLine());
```

Here, the value is entered as *Console.ReadLine()* and the result is a *string* which must be converted to an *int.* You can do that with the method *Parse()* which is a method that can convert a *string* to an *int*, if the string contains a legal value. In this case the value is stored in the variable *a*. To make that possible the result from *Console.ReadLine()* must be a legal integer. Is it not the case (because the user has entered an illegal value), then the conversion will raise an exception, which means that the conversion is interrupted with an error. When you have code that can raise an exception, the code can be placed in a *try* block. If so and there is an exception the runtime system interrupts the *try* block, and the control is transferred to the subsequent *catch* block. In this case, it means that if one of the three entered values can not be converted to an *int*, then the program jumps to the *catch* part and prints an error message.

If the three arguments can be converted correctly the result is printed as follows:

```
a + " + " + b + " + " + c + " = " + (a + b + c)
```

This is a relatively complex expression, and may not be easy to read, but to understand the expression, it is easiest to compare with the result:

```
13 + 17 + 19 = 49
```

The numbers 13, 17 and 19 are the values of the three variables *a*, *b* and *c*. The + signs in quotations is just text and are included in the result as strings. If you for example consider

```
a + " + "
```

the + sign means string concatenation of the value of the variable *a* and the value of the text " + ", which is a + sign with a gap on both sides. The last parentheses in the result expression

```
(a + b + c)
```

calculates the sum of the three variables, and from that the value 49.

Exercise 2: Sum of four numbers

You should write a program similar to the above, but there must be the following differences:

1. The program should calculate the sum of four numbers instead of three numbers
2. The entered numbers must be stored in *string* variables before they are converted
3. The numbers should this time be decimal numbers instead of integers, that is the type must be *double*

Above is shown how the method *int.Parse()* can convert a text to an *int*. To convert a text into a *double*, you can correspondingly apply the method *double.Parse()*.

An example of running the program could be:

```
Enter the 1. number: 3.25
Enter the 2. number: 1.75
Enter the 3. number: 9.85
Enter the 4. number: 2.55
3.25 + 1.75 + 9.85 + 2.55 = 17.4
```

Note especially that the decimal point is dot.

4.1 OPERATORS

The operators are special symbols known by the compiler, that perform specific operations on one, two or three operands, which are typically values or variables. Above I have already used the + operator, as an operator between strings that act as string concatenation, while the operator between numbers is interpreted like addition of numbers. Also a declaration in which a variable is assigned a value contains an operator, as the equal symbol is an operator.

The operators have different priorities, which affect the order in which they are evaluated if an expression contains multiple operators, and the following table shows C#'s operators in order of priority sorted in descending order. If an expression contains multiple operators, the operators with the highest priority are evaluated first. Operators with the same priority are evaluated from left to right. However, that does not apply to the assignment operators, which is evaluated from right to left.

() . [] function(...) new typeof sizeof checked unchecked
+ - ! ~ ++ -- (unary operatorer)
* / %
+ -
<< >>
< > <= >= is as
== !=
&
^
|
&&
||
?:
= *= /= %= += -= <<= >>= &= ^= |=

Many of these operators require no special explanation, and others, I first will deal with later, but below are a few comments on some individual operators.

The assignment operator

Note first that assignment is an operator, for example

```
int a = 2;
a = 3;
```

It is undisputed the most commonly used operator.

The arithmetic operators

Multiplicative and additive operators generally referred to as arithmetic operators and are the operators for the four arithmetical calculation operations. Furthermore, there is the modulus operator, which is closely linked to division. All five operators have two arguments. If the arguments are integers, the result is again an integer. This means the division is integer division. That is

```
23 / 7 = 3
```

as the division goes a 3 time and the rest is thrown away. Modulus is the rest with division, and such is

```
23 % 7 = 2
```

when 23 divided by 7 gives rest 2. Look at the following statements

```
Console.WriteLine(23 / 7);
Console.WriteLine(23 % 7);
Console.WriteLine(-23 / 7);
Console.WriteLine(-23 % 7);
Console.WriteLine(23 / -7);
Console.WriteLine(23 % -7);
Console.WriteLine(-23 / -7);
Console.WriteLine(-23 % -7);
```

They will print

```
3
2
-3
-2
-3
2
3
-2
```

Here, the sign is not quite obvious, but for two integers *a* and *b* are

```
a % b = a - round(a / b) * b
```

where *round(a / b)* means *a / b* with any decimals thrown away. From this formula, it is easy to figure out that the above results are correct. The modulus operator can also be used if the arguments are floating-point numbers, and the formula is the same, but the result has rare interest in this case. The arithmetic operators can be combined with the assignment operator. For example means

```
a += 2;
```

the same as

```
a = a + 2;
```

Although it is not an arithmetic operator, you should note that the + operator is is also used for strings, and for example means

```
String s = "Hello" + "World";
```

string concatenation and the value of the variable *s* is *HelloWorld*. This means that if the type of at least one of the operands are *string* the operator + means string concatenation and not addition.

Unary operators

There is a family of operators which takes only one argument, and that includes the operators to sign, negation and the two special operators ++ and --. Finally there is also the operator for binary complement, which I will not mention in this place. A common feature of these operators is that it is the operators with the highest priority. Consider as an example the following statements:

```
int a = 2;
Console.WriteLine(++a);
Console.WriteLine(a++);
Console.WriteLine(a);
```

If performed, you get the result:

```
3
3
4
```

The variable *a* has the value 2. Executes ++*a*, it means that the value of the variable is incremented by 1, after which the value is printed. The result is therefore 3 (the variable *a* has the value 3). Executes then *a*++, this means that the value of *a* is printed, after which the variable *a* is incremented by 1. The statement therefore prints 3, but after the statement is executed, the value of *a* is 4. This shows the last *WriteLine()* statement. The -- operator works similar but decrements the value of a variable by 1.

With regard of negation you can consider the statements:

```
int a = 2;
int b = 3;
Console.WriteLine(a == b);
Console.WriteLine(!(a == b));
```

If the statements executes you get

```
false
true
```

a == b is a condition, and the first *Console.WriteLine* statement prints the value of this condition, which is *false*. The next statement prints the negation of the condition that is *true*. You should note the parentheses in the condition *!(a == b)*. They are necessary because ! has higher priority than ==.

Operators to conditions

It includes the relational operators, comparison operators and the logical operators. Generally, these operators are not the big challenges, but you should be aware of the logical operators. Consider the following statements

```
int a = 2;
int b = 3;
int c = 2;
Console.WriteLine(a == b || a == c);
Console.WriteLine(a == b && a == c);
Console.WriteLine(a == b || a == c && a == c);
```

that prints

```
true
false
true
```

It is not so very strange, but note the last, where the *&&* operator is performed before || because of priorities.

The question operator

As the last operator I will mention the question operator, which is the only operator that takes three arguments. It is of the form

condition ? arg1 : arg2;

Here are *arg1* and *arg2* expressions of the same type and the result of the operator is the value of *arg1*, if the *condition* is *true*, and otherwise the value is *arg2*. Consider the following statements:

```
int a = 23;
int b = 17;
Console.WriteLine(a < b ? a : b);
```

They prints

```
17
```

First the *WriteLine()* statement evaluate the condition

```
a < b
```

Because it is *false*, the statement prints the value of *b,* that is the smaller of the two numbers *a* and *b*.

Exercise 3: Operations

Try to determine what the following statements prints, with paper and pencil and without writing a similar program:

```
int a = 2;
int b = 3;
Console.WriteLine(a + b / 2 == 2);
Console.WriteLine((a + b) / 2 == 2);
Console.WriteLine(a / b);
Console.WriteLine(b / a);
Console.WriteLine(a % b);
Console.WriteLine(b % a);
Console.WriteLine(a > b);
Console.WriteLine(a != b);
Console.WriteLine(a < b ? "Gudrun" : "Olga");
Console.WriteLine(a > b ? "Gudrun" : "Olga");
Console.WriteLine(b == a++);
Console.WriteLine(--b == --a);
Console.WriteLine(a == b++ && b == 3);
Console.WriteLine(++a == b++ && a < b && a % 2 > 0);
```

When you are finished, you can check the result by writing a program that has a *main()* method with the above statements.

Exercise 4: Arithmetic

Write a program where the user must enter two integers. The program should then print

- the sum of the numbers
- the numbers difference
- the numbers product
- the numbers quotient
- the numbers modulus

The result could, for instance be:

```
Enter the 1. number: 123
Enter the 2. number: 13
Sum:         136
Difference:  110
Product:     1599
Quotient:    9
Modulus:     6
```

4.2 LITERALS

Primitive types are built into the language, and variables of primitive types can be assigned a value using literals, for example

```
bool res = true;
char ch = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
string s = "Hello World";
```

where the last example shows a literal, but not for a primitive type.

There are some syntax for how to define literals, and below are the most important.

Literals for integers

Generally, you specify an integer using the digits 0-9, and thus a number from the 10-number system. An integer is interpreted as *long* if it ends with the letter L or l. Otherwise, the type is *int*. It is recommended that you use a large L, as a small l is hard to distinguish from the digit 1. You can also ends an integer with U or u which means unsigned.

If an integer should be interpreted in hexadecimal you must start the number with 0x, and the symbols are the digits and the letters A, B, C, D, E and F. Finally, you can specify that the integer should be interpreted binary starting the integer with 0b and then the symbols 0 and 1 (binary numbers and hexadecimal numbers are explained later).

Examples could be the following:

```
int t1 = 123;
int t2 = -123;
uint t3 = 123U;
long t4 = 123L;
ulong t5 = 123UL;
int t6 = 0x123;
ulong t7 = 0xABCUL;
int t8 = 0b101;
```

Literals defining floating points

A floating point or decimal number is interpreted as the type *float* if it ends with the letter F or f. Otherwise, its type is *double*, which can also be set explicitly with the letter D or d. You can also specify an exponent with the letter E or e. The following variables have all the value 1234.56:

```
double x1 = 1234.56;
double x2 = 1.23456E3;
float x3 = 1234.56F;
float x4 = 12345.6E-1F;
```

Here you need specifically to note that 1.23456E3 means 1.23456 times 10 in the third.

Literals defining characters and strings

Literals defining the types *char* and *string* can contain any Unicode (UTF-16) characters, and to the extent that the characters are present, they can be used directly. Is a character not on your keyboard, it can be added as *Unicode escape sequences* like '\u0108'. For *char* literals use single quotes, while for *string* literals use double quotes:

```
char c = 't';
String s = "Gorm den Gamle";
```

C# also supports single escape sequences for special control characters:

- \b backspace
- \t tabulator
- \n line feed
- \f form feed
- \r carriage return
- \" double quote
- \' single quote
- \\ backslash

Finally, you can prefix a string with the character @, which means that escape sequences should not be interpreted. Consider as an example the following statements:

```
string s2 = "1234567890";
char c2 = '\u0108';
char c3 = '\'';
string s3 = "abc\tde\\\"fg";
string s4 = "a\nb\nc";
string s5 = "\u0041\u0042\u0043";
string s6 = @"1234567890";
string s7 = @"a\nb\nc";
string s8 = @"\u0041\u0042\u0043";
```

Use of underscores in number literals

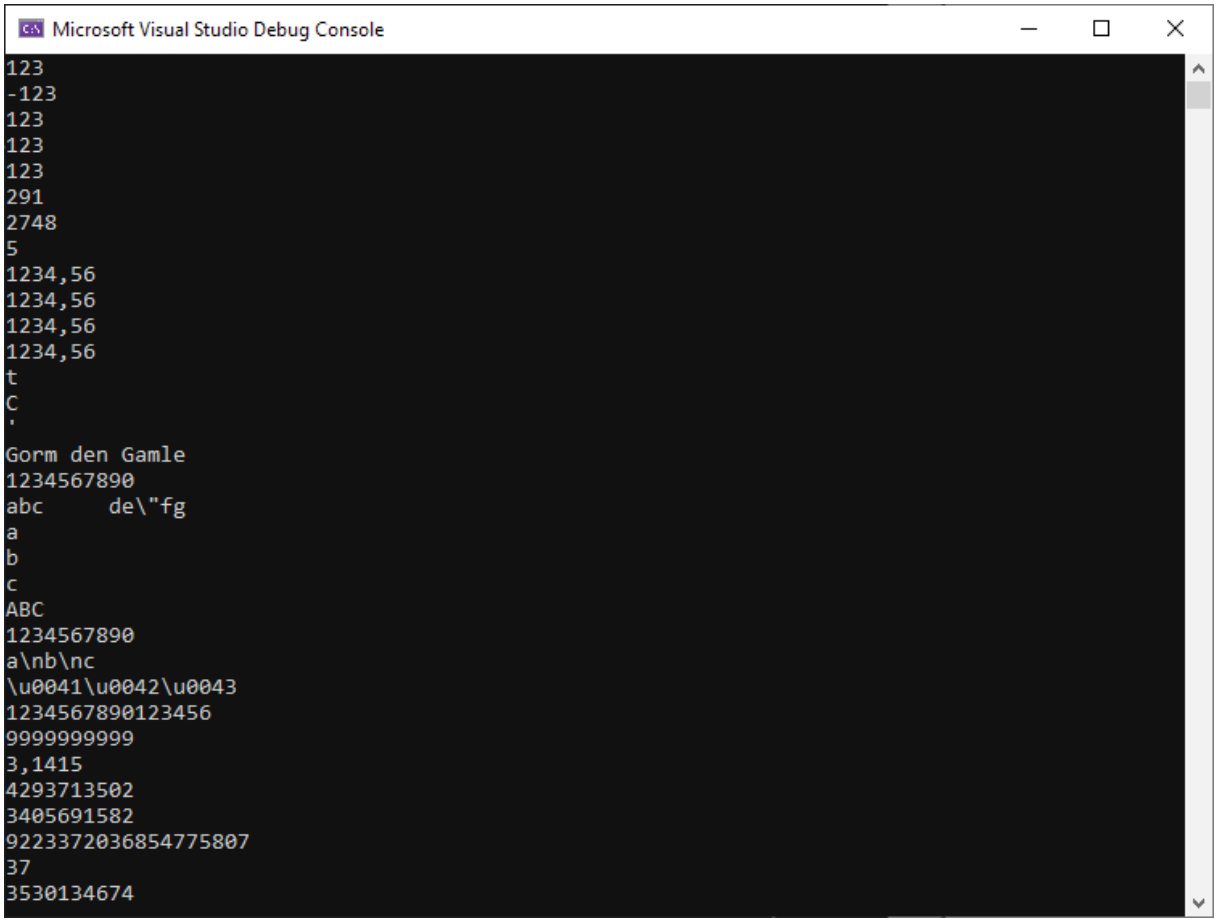
As a final note regarding literals, I will mention using '_' in literals for numbers. As examples it is legal to write

```
long cardId = 1234_5678_9012_3456L;
long cprnr = 999999_9999L;
float pi = 3.14_15F;
long ip = 0xFF_EC_DE_5E;
long words = 0xCAFE_BABE;
long max = 0x7fff_ffff_ffff_ffffL;
byte val = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

and all occurrences of '_' are ignored. The aim is only to enhance the readability of large numbers. There are a few limitations:

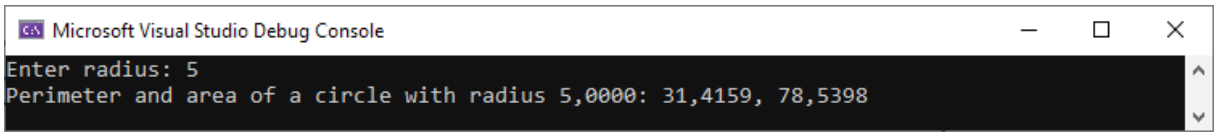
- A number must not start or end with the character _
- The character _ must not stand next to the decimal point in a floating point number
- The character _ must not stand before an F or L

The above examples of literals are from the example *Literals* and if you run the program the result is:



4.3 FORMATTING OUTPUT

In this section I will look at how the user can format the program's output. As an example I will show an application, where the user must enter the radius of a circle, and the program calculates and writes the circle's circumference and area. Below is an example of a running program:



The code can be written as follows (where I have not included the program's *using* statements and the program's namespace):

```
static void Main(string[] args)
{
  Console.Write("Enter radius: ");
  string text = Console.ReadLine();
  double r = Convert.ToDouble(text);
  double p = r * 2 * Math.PI;
  double a = r * r * Math.PI;
  Console.WriteLine(
    "Perimeter and area of a circle with radius
    {0:F4}: {1:F4}, {2:F4}", r, p, a);
}
```

The program has in principle the same structure as the other examples and consists only of a *Main()* method. The program writes a help text, then the user must enter a number (radius):

```
string text = Console.ReadLine();
```

The entries are returned as a string, that is to a variable of the type *string*. *ReadLine()* always returns a *string*, and it is then the program's task to convert the input to a different type as needed. In this case, the input is converted to a *double* with the statement:

```
double r = Convert.ToDouble(text);
```

Convert is a class in the namespace *System* which defines a family of conversion functions. Note that these, here *ToDouble()*, requires that the user has actually entered a legitimate number. If not, the program stops with an exception. Next the perimeter and area are calculated:

```
double p = r * 2 * Math.PI;
double a = r * r * Math.PI;
```

Here *Math.PI* is a constant in the class *Math* which is a class in the *System* namespace. Finally the program write the result with *WriteLine()*, but this time the function has several arguments. The first argument is called a format string and is followed by three variables. The values of the variables are inserted into the format string determined by the so-called placeholders. For example is {0: F4} a placeholder that indicates that here, the first variable after the format string should be inserted, that is the value of the variable *r*. The next placeholder is called {1: F4}. It indicates that here the variable *p* is inserted, the variable number 2 after the format string. F4 means that the value is added as a decimal number (F) with 4 decimals after the decimal point. Similarly, states {2: F4} to be inserted a value formatted as a decimal number with 4 digits. It is in this case, the variable *a*.

There are following options to format a placeholder:

C	Currency (depends on the local setting)
D	Integer
E	Exponential form (float, double)
F	Fixed decimal (float, double)
G	General (F or E)
N	Numeric with thousands
X	Hexadecimal

The next example performs a calculation, where the user must enter the unit price and number of units of an item. Then the program calculates the total price excl. VAT, VAT, total price incl. VAT and writes the result on the screen:

```
static void Main(string[] args)
{
  Console.Write("Enter the unit price: ");
  string text = Console.ReadLine();
  double price = Convert.ToDouble(text);
  Console.Write("Enter the number of units: ");
  text = Console.ReadLine();
  int quantity = Convert.ToInt32(text);
  double amount = price * quantity;
  double vat = amount * 0.25;
  double total = amount + vat;
  Console.WriteLine("{0, -15} {1, 10:F}", "Unit price", price);
  Console.WriteLine("{0, -15} {1, 10:D}",
  "Number of units", quantity);
  Console.WriteLine("{0, -15} {1, 10:F}", "Total excl. VAT", amount);
  Console.WriteLine("{0, -15} {1, 10:F}", "VAT", vat);
  Console.WriteLine("{0, -15} {1, 10:F}", "Total incl. VAT", total);
}
```

If you run the program, the result could be the following:

The program works just like the previous program, only this time you must enter two values. Moreover, the placeholders are more complex. If for example you look at the statement:

```
Console.WriteLine("{0, -15} {1, 10:F}", "Unit price", price);
```

there are two placeholders. {0, -15} is the first, and insert the words "Unit price". -15 means that the field is 15 characters wide, and when the number is negative, the value must be left justified. Note that there is no format character, and then it is the data type of the element that determines the format type. The next placeholder {1, 10:F} means that the next item to be formatted must be right-justified in a field of 10 characters and as a decimal number. As the number of decimal places is not specified the default value is used, which is 2.

The last example in this section shows how to formats of the result of the type *DateTime* which is a class representing a date and time:

```
static void Main(string[] args)
{
  DateTime dt = DateTime.Now;
  Time1(dt);
  Time2(dt);
}

static void Time1(DateTime t)
{
  Console.WriteLine("{0:D2} {1:D2} {2:D2} {3:D3}",
    t.Hour, t.Minute, t.Second, t.Millisecond);
}

static void Time2(DateTime t)
{
  Console.WriteLine(t.ToLongDateString());
  Console.WriteLine(t.ToLongTimeString());
}
```

In *Main()* the machine clock is read and its value is stored in a variable:

```
DateTime dt = DateTime.Now;
```

DateTime is a class that contains a number of methods to date and time. *Now* is a property, which always contains the current value of the hardware clock. This value is stored in a variable called *dt* which type is *DateTime*. Next, a method *Time1()* is called, where the variable *dt* is sent as a parameter. This means that the value is known and can be used in the method *Time1()* with the name *t*. The method prints the time in terms of hours, minutes, seconds and milliseconds, each part separated by spaces. You should note the placeholders. For example means {0: D2} that the first variable to be formatted in a field as an integer, and that the field should be two characters in order to insert a leading 0, if there is only one digit. You will also notice how you refer to the values. *t* is a variable whose type is *DateTime*, which is a class. The class defines a number of properties, for example *t.Hour* for hours, and you refers to the individual characteristics by the variables name followed by a period and the feature name.

The class *DateTime* provides other opportunities. The method *Time2()* writes the current date and time, but here I used methods from the *DateTime* class that formats the result as a string.

If you run the program the result could be:

DateTime is a class (actually a *struct*), and as you can see, there are some new things such as properties etc.. All this will be dealt with in detail later. The same applies to methods and parameters that are also used in this example without going into detail on the meaning, but you can think of a parameter as a variable that specifies the value a method has to work on. In the examples here, the methods use the value of the variable *dt*, but it exists and is only known in *Main()* and is thus not known in the other two methods. We need a mechanism that can transfer *dt*, when the methods are called, and that's exactly what parameters are used to.

You should note that in this example there is no particular reason for writing the program using several methods, the aim is also to show the syntax alone, but the use of methods is a central issue for the following.

4.4 OBJECTS

C# is an *object oriented programming* language, and there are some basic concepts needed to know, before you can write programs that perform something interesting, and the following is an overall introduction to objects, classes, inheritance, interfaces and namespaces, where I primarily will focus on how these concepts relate to the real world, while providing a basic introduction to the C# syntax.

An object is a software module having a state and a behavior. Software objects are often used to model real-world objects. This section explains how an object represents state and behavior and introduces the concept of data encapsulation. The important thing is to explain the advantages of designing software in this way as a family of cooperating objects.

Objects are the key to understanding the object-oriented technology. If you look around, you will find many examples of real-world objects. It could be the same as a desk, your wife, a particular colleague, a customer, your television, your car and so on. These objects

are characterized by two things: They all have state and behavior. A customer has a name, an address, a phone number, a balance and so on. It is the customer's state. The customer can order a product, the customer can return a product, the customer can pay for a product. It is the customer's behavior. In the same way a car's state may be make, color, power, etc., and its behavior might be starting, changing gears, braking and so on. By in this way to identify the state and behavior of objects in the real world, you have a good starting to think object-oriented, and thus to find the objects in the development of a program.

Software objects are conceptual, but else looks like the real world objects. They also have a state and related behaviors. An object stores its state in variables or fields, and it performs its behavior through methods. Methods acting on an object's internal state and serves as the primary mechanism for object-to-object communication. The hiding of the objects internal state and require that all interaction must be made using the objects methods is called for data encapsulation and is a fundamental principle of object-oriented programming. With data encapsulation it is the object, which has control over how its state changes as it is the object that through its methods determines what the outside world can do with it.

Above I have dealt with variables in C#, but variables can be several things, and there are basically the following opportunities:

1. *Instance variables* (non-static fields) as objects uses to store their state. All variables that in a class are defined as non-static, are instance variables, and each object of the class has its own instance variables, as a specific object's state is independent of all other objects state, even if it is objects of the same class.
2. *Class variables* (static fields) are variables defined *static*. Aside from that they are created in the same way as instance variables, there are only created one copy of these variables, and all objects of a certain class will share all class variables. A static variable is thus designed to store values to be used by all objects of a class.
3. *Local variables* are in principle created in the same way as instance variables, but they are created in a method, and is known only to this method. All variables that I've used above, are local variables. They are typically used to store temporary values that a method needs to do its job. Instance variables and class variables have a default value if they are not initialized by the programmer. In contrast, local variables must be initialized with a value. A local variable is created when the method in which it is defined, is called, and the variable is automatically deleted again when the method is completed.
4. *Parameters* are variables that you specify in parentheses after the name of a method. Parameters seems a bit like local variables, but they are used to pass values to a method when it is called. Thus, you can think of parameters as variables that defines the values that a method should act on. *Main()* has a parameter called *args* that is used to transfer arguments to the program on the command line.

To construct program code as a family of software objects provides a number of advantages:

- Modular code, where the code for an object can be written and maintained independently of the code to other objects.
- Information hiding, where all interaction with an object is by means of its methods. The details concerning state of the object and the internal implementation remains hidden from the outside world.
- Code reuse, where already developed objects (perhaps written by another software developer) can be used in the current program. It allows the objects to be developed and tested by specialists, which you can trust on and use them in your own programs.
- Modifiable, where a particular object without affecting the rest of the program can be replaced with another, if the object for some reason is found inexpedient.

In the real world, you often meet many individual objects that are similar and of the same kind. There are thousands of cars, all of the same make and model and is constructed in exactly the same way and of identical components. In the object-oriented world wee say that a particular car is an instance of the class cars defining a particular make and model. A class is a description of how specific cars behave.

Wee use the same principle in relation to software. Assume that a program need objects for banknotes. A banknote is characterized by a value which is the banknote's state, and the only thing you have to do with a banknote is to read its value and know how the note looks so you can distinguish it from other notes. Accordingly, you can define the concept of a banknote as follows:

```
using System;

namespace BankProgram
{
  class BankNote
  {
    private int value;

    public BankNote(int value)
    {
      this.value = value;
    }

    public int Value
    {
      get
      {
        return value;
      }
    }
  }
}
```

It is an example of a C# *class*, which here has the name *BankNote*. A definite note has as mentioned a value that is the note's state, and this value must be stored somewhere. To this end, the class defines a variable called *value*. A variable has a type, here an *int* and means that the variable may contain an integer. The variable *value* is an example of an instance variable. The variable is defined *private*, which means that it can only be referenced from the class itself, and that is what I have called data encapsulation. The class should be used for something, and you should be able to read the note's value. The class has a property named *Value*, whose value is the banknote's value that is the value of the variable *value*. This property defines the objects behavior. Note that the property has a type, which here is *int*. Wee say that the property returns a value. The property defines one method called *get* and is the method which returns the value of the variable *value*. A property is used to make the value of an instance variable available to those using an object of the class. The name of the property is *Value* and then the same name as the variable, but starting with an upper case letter. It is not a requirement, but it is common practice. The class has another method that has the same name as the class. It is a special method, which does not define a behavior of an object, but is used to initialize an object's state when creating a new object. Such a method is called a *constructor*.

You should note that the class has no *main()* method, because the class is not a program, but merely defines a concept that can be used by any application.

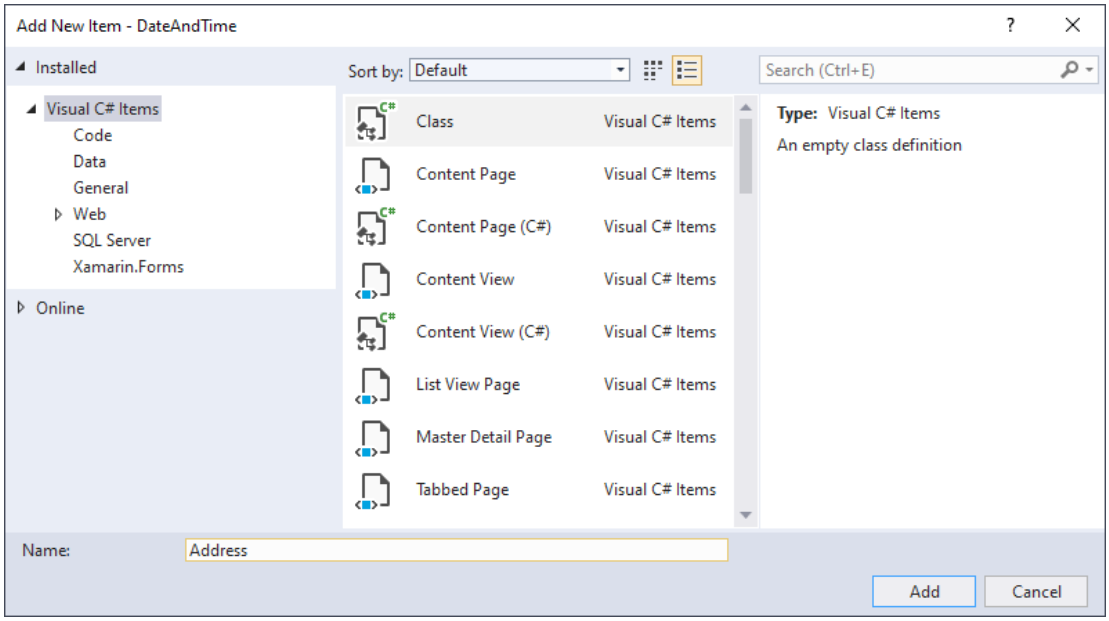
With the class *BankNote* available, a program can create *BankNote* objects and do something with them. Below is a *main()* method from a program that creates two *BankNote* objects. The first has the value of 100, while the next has the value of 200:

```
namespace BankProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      BankNote note1 = new BankNote(100);
      BankNote note2 = new BankNote(200);
      Console.WriteLine(note1.Value);
      Console.WriteLine(note2.Value);
    }
  }
}
```

You must specifically note how to create an object. An object has a name, for example *note1*, which is a variable. The object also has a type, which is here *BankNote*, and when the type is a *class* rather than a primitive type, the object must be created with the *new* operator. Here you specify the value of the banknote, and exactly it means that the class's constructor is executed and transmits the value to the object's instance variable *value*. If the *Main()* method is performed it prints the values of the two *BankNote* objects. For that to be possible, you have to refer to the *BankNote* object's value. It is not possible because of data encapsulation, but a *BankNote* object has a behavior in terms of the property *Value*, so you can refer to the value.

4.5 THEADDRESS

As another example of using objects and classes, I will show another version of the program *InputAddress*. The program must do exactly the same as in the previous example, but the code must be written in a different way, where an address this time should be represented by an object whose type is a class *Address*. I have created a new project, which I have called *TheAddress*. Next, in the menu under *Project* I have selected *Add Class...* and in the following window i have selected *Class* and entered the name *Address*:



When I click on *Add* Visual Studio adds a class with the name *Address* to the project:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace TheAddress
{
  class Address
  {
  }
}
```

Visual Studio creates a skeleton for a class and three *using* statements, where only the first is necessary. The other two can be deleted if you do not want them to be there, but it does not matter to the finished program.

Then the code must be written. The class must have four variables for the properties that an address should have. They are instance variables and for data encapsulation they are all defined *private*. This means that the class's variables can only be referenced by the class's own methods, and thus it is the programmer that has to open up for the variables to be used by other objects. In order for the class to be used for something, it is usually necessary to read the values of the variables, and it happens by properties as shown in the class *BankNote*. A property has a *get* method, which returns the value, but if it also should be possible to change the value of a variable, you must define a *set* method that has the new value of the variable. With these considerations, the class can be written as follows:

```
using System;

namespace TheAddress
{
  class Address
  {
    private String name;
    private String addr;
    private String code;
    private String city;

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

    public string Addr
    {
      get { return addr; }
      set { addr = value; }
    }

    public string Code
    {
      get { return code; }
      set { code = value; }
    }

    public string City
    {
      get { return city; }
      set { city = value; }
    }

    public void Print()
    {
      Console.WriteLine();
      Console.WriteLine("Address:");
      Console.WriteLine();
      Console.WriteLine(" Name:      " + name);
      Console.WriteLine(" Address:   " + addr);
      Console.WriteLine("            " + code + " " + city);
    }
  }
}
```

Here you should notice three things. First, the class, unlike the class *BankNote*, has no constructor. It is not absolutely necessary for a class to have a constructor, but will typically be the case. In this case, the variables are not initialized, and they are assigned default values, as here is *null*, which simply means that the variables have no value. Secondly, note that the properties has set methods and notes the syntax and the value to be assigned to a variable always is referenced as *value*. Finally there is the method *Print()* that prints the contents of the four instance variables on the screen. Note that the code is the same as that used in the application *InputAddress*.

With the class *Address* available, the program can then be written as follows:

```
using System;

namespace TheAddress
{
  class Program
  {
    static void Main(String[] args)
    {
      Address ad = EnterAddress();
      Console.WriteLine();
      ad.Print();
    }

    private static Address EnterAddress()
    {
      Address address = new Address();
      Console.Write("Enter name: ");
      address.Name = Console.ReadLine();
      Console.Write("Enter address: ");
      address.Addr = Console.ReadLine();
      Console.Write("Enter zipcode: ");
      address.Code = Console.ReadLine();
      Console.Write("Enter town: ");
      address.City = Console.ReadLine();
      return address;
    }
  }
}
```

and the program then works in exactly the same way as the application *InputAddress*. It is not clear what the advantage of this solution is, as the code fills far more than before, and seen solely in relation to this program, there is no benefit either, but as programs grow bigger and more complex, the division into classes are completely fundamental to the program architecture and for the maintenance of the program. The class *Address* represents a subject in the program's problem area. Everything that concerns an address, especially the data presentation in the form of variables, has been moved to its own class. It is then the class's methods that determines what is possible with an object of the class (an object is a specific address). You can then think of an *Address* as a type that defines a concept (a kind of objects) that the program needs, and then the program can create objects of that type as needed. In the program above, one object, called *ad*, is created. In that way to move everything about an address into its own class, you get a split of the code into parts where each part (*class*) has its own well-defined task and responsibility. It is a step towards modular programming.

Problem 3: TheBook

You must write a program that solves exactly the same task as problem 2, that is where you can enter information about a borrower and a book, and the program must then print the same recall for the book. However, the program must be written after the same pattern as the example *TheAddress*.

Start with a new project, which you can call *TheBook*. Then add a class named *Borrower*, that should represents a borrower using three variables. Unlike the class *Address* in the previous example, there should be no *print()* method, but only properties for the three variables. Add a correspondingly class *Book* which represents a book using two variables when this class also should not have any *print()* method, but properties for the two variables.

The program must then have the following architecture:

```
using System;

namespace TheBook
{
  class Program
  {
    static void Main(String[] args)
    {
      Borrower borrower = EnterBorrower();
      Book book = EnterBook();
      print(borrower, book);
    }

    private static Borrower EnterBorrower()
    {
      // enter information for the borrower
    }

    private static Book EnterBook()
    {
      // enter information for the book
    }

    private static void print(Borrower borrower, Book book)
    {
      // print the recall
    }
  }
}
```

4.6 INHERITANCE

Different kinds of objects often have several characteristics in common with each other. If you think of banknotes they all have a value, but they differ with respect to how they look. They have different colors, and they show different pictures. In object-oriented programming it is possible that classes can inherit each other. That means that you can collect common properties in a base class, while the differences can be placed in derived classes. Below is a class that represent a note with the value 100 (a Danish banknote):

```
using System;

namespace BankProgram
{
  class BankNote100 : BankNote
  {
    public BankNote100() : base(100)
    {
    }

    public void Print()
    {
      Console.WriteLine("Den gamle Lillebæltsbro og Hindsgavl-dolken");
    }
  }
}
```

The class is called *BankNote100*, and after the class name you define with a colon that the class inherits the class *BankNote*. This means that the class inherits all the properties and methods that a *BankNote* have and is possible expanded with new properties and methods. In this case *BankNote100* extends *BankNote* with a method *Print()*, to simulate how the note looks (a text written in Danish). The class *BankNote* has a constructor which initialize the instance variable *value*. A *BankNote100* must also initialize this variable (with the value 100). It happens in the class's constructor with a call of *base(100)*, which means that the base class's constructor is executed.

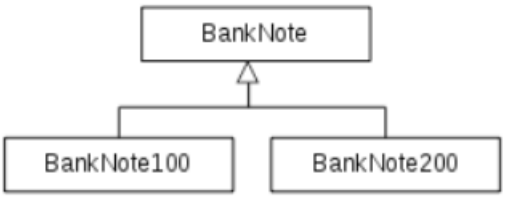
It is clear that in quite the same way you can write a class *BankNote200* representing a banknote with the value 200.

With these classes available you can write the following *Main()* method:

```
static void Main(string[] args)
{
  BankNote100 note1 = new BankNote100();
  BankNote200 note2 = new BankNote200();
  note1.Print();
  note2.Print();
  Console.WriteLine(note1.Value + note2.Value);
}
```

Here, you create two objects of the types respectively *BankNote100* and *BankNote200*. Next the objects *Print()* method are called. Note the syntax, and how to perform a method on an object using the dot operator. The last statement prints the sum of the two objects values. Here you should note the use of the property *Value*. It is defined in the class *BankNote*, but is inherited to the classes *BankNote100* and *BankNote200*.

Often wee illustrate inheritance as follows:



In this context the base class *BankNote* is also called a super-class while the derived classes *BankNote100* and *BankNote200* are called sub-classes. Sometimes wee also say that *BankNote* is a generalization of *BankNote100* and *BankNote200* while these classes are called specializations of *BankNote*.

Interfaces

An object's properties and methods are known to the outside world through the *public* methods that the class defines and thus made available as services. Methods form the object's interface to the outside world. In order to define an object's characteristics, one can use an *interface* which defines the methods of an object of a certain class. For example you can define a banknote as follows:

```
namespace BankProgram
{
  interface Note
  {
    public int Value { get; }
    public void Print();
  }
}
```

It looks like the definition of a class, but you should notice that it simply is only a definition and that the interface does not contain code that performs something. Note the syntax for a property and how to define that the property *Value* only should have a get method. Also note that the interface has no *using* statements. This will often be the case, but in this case there are no references to .NET types which are not known and are therefore unnecessary.

Given an interface you can define that a class must implement this interface. Thus you specify that the class has the methods that the interface defines. For example the class *BankNote* can implement the interface *Note* as follows:

```
namespace BankProgram
{
  abstract class BankNote : Note
  {
    private int value;

    public BankNote(int value)
    {
      this.value = value;
    }

    public int Value
    {
      get
      {
        return value;
      }
    }

    public abstract void Print();
  }
}
```

Note first how to define that the class implements an interface, which is the same syntax as for inheritance. This means that the class should have a property *Value* and a method *Print()* - it must implement the methods that the interface defines. It is not the case, since it do not implements the method *Print()*. It can not do that because it does not know how this method should be written. It is the derived classes *BankNote100* and *BankNote200*, that has that knowledge. The problem is solved by defining the class as *abstract* and the method *Print()* must in the class *BankNote* be defined as *abstract*. The method then has to be implemented in the derived classes, what it indeed is. That a class is *abstract* has further that consequence that you can not instantiate objects whose type is the abstract class, in this case *BankNote*. It is also reasonable, else you could instantiate notes with an arbitrary value, which does not model the real world in a meaningful way.

The class *BankNote100* inherits *BankNote* which is an abstract class, and *BankNote100* must then implements the abstract method *Print()* what it already do, but it is necessary with the word *override* to indicate that it is a method overridden from the base class:

```
public override void Print()
{
  Console.WriteLine("Den gamle Lillebæltsbro og Hindsgavl-dolken");
}
```

The same goes for the class *BankNote200*. Below is a new version of the *main()* method:

```
static void Main(string[] args)
{
  Note note1 = new BankNote100();
  Note note2 = new BankNote200();
  note1.Print();
  note2.Print();
  Console.WriteLine(note1.Value + note2.Value);
}
```

There will again be created two objects, respectively a *BankNote100* and a *BankNote200* object, but their type is *Note*. A *BankNote100* is especially a *BankNote*, which is again is a *Note*. This means that the two objects in the rest of the program alone is known from the defining interface and then in the rest of the program are objects that provides two services *Value* and *Print()*.

From the above, it is not obvious what the advantage of interfaces are and it will first be clear later, but the explanation must be sought in that, *note1* and *note2* are in the program only known from the defining interface, which means that the two objects are treated equally independent their specific types.

The project *BankProgram* implements the above classes and interface.

4.7 NAMESPACES

A namespace organizes a family of related classes and interfaces and also other types. Conceptually, you can think of a namespace in the same way as a folder in which a namespace contains classes that somehow belong together. Since programs written in C# may consist of hundreds or thousands of individual classes, it makes sense to keep things separated by placing related classes and interfaces in namespaces.

When Visual Studio creates a project, it also creates a namespace and place the project's classes and interfaces in that namespace, for example

```
namespace BankProgram
{
  interface Note
  {
    public int Value { get; }
    public void Print();
  }
}
```

which means that the interface is in the namespace *BankProgram*. So long that the project contains only a single namespace, it is not something you need to think about, but it means exactly that the name of the type *Note* is

```
BankProgram.Note
```

A *namespace* statement defines the namespace to which a type belongs.

The .NET platform includes a huge library (a collection of namespaces), which contains components, the programmer may use in his own applications. This library is known as the .NET *API*. Its namespaces represents the tasks most often associated with general issues within the program development, and examples are namespaces for developing applications with a graphical user interface.

To refer to classes other than in the defining namespace, the namespace must be referenced with a *using* statement. The following statement

```
Concole.WriteLine("100 kr., ");
```

prints a text, and *Console* is actually the name of a class that exists in the namespace *System*. This namespace contains all of the most common classes from the .NET API.

Exercise 5: More banknotes

Create a copy of the project *BankProgram*. In the class *Banknote100*, you should modify the *Print()* method as follows:

```
public void Print()
{
  Console.Write("100 kr., ");
  Console.WriteLine("Den gamle Lillebæltsbro og Hindsgavl-dolken");
}
```

Test the program and note that the text of both statements is on the same line.

Modify the *Print()* method in the class *BankNote200* in the same way.

Write three classes *BankNote50*, *BankNote500* and *BankNote1000* representing banknotes with values respectively 50, 500 and 1000. The three classes are of course similar to the classes *BankNote100* and *BankNote200,* but the value is not the same and the text in the method *Print()* is different. The classes represents Danish banknotes, and the notes text are respectively

1. Sallingsundbroen og Skarpsalling-karret
2. Dronning Alexandrines bro og bronzespanden fra Keldby
3. Storebæltsbroen og Solvognen

In *Main()*, create an object of each of the five specific banknote types. You should print the five objects and the sum of the five banknotes values.

Problem 4: Employees

You should write a program that creates objects that represents employees in a company. You should solve the problem by following the steps below.

1.

Create a new project in Visual Studio, which you can call *Employees*. Add the following interface to the project:

```
namespace Employees
{
  // Interface that defines an employee of a company
  interface Employee
  {
    public string Name { get; } // returns the employee's name
    public int Salary { get; }  // returns the employee's
                                   monthly salary
    public void Print();        // prints
information about the employee
  }
}
```

This interface defines what an employee is and in this exercise an employee do not have any other characteristics.

2.

Add an *abstract* class *AbstractEmployee*, which implements the interface *Employee* and represents an employee when

1. the class should have two variables, for respectively the name and the monthly salary
2. the class should have a constructor that initializes the two variables
3. the class should implement the two properties *Name* and *Salary*

When you implements the property *Salary* you must define it *virtual*:

```
public virtual int Salary
```

which means the property can be overridden in a derived class.

3.

Add a class *Bookkeeper* which inherit *AbstractEmployee*. Besides the constructor (which must have two arguments) the class must implement the method *Print()*, that prints the name, the job title and monthly salary. The result could, for instance be:

```
########################################
Gudrun Jensen, Bookkeeper
Monthly salary: 45000
########################################
```

where the name and monthly salary are the values passed to the constructor.

4. Add a corresponding class *Janitor* that represents a janitor. This class is basically identical to the above but the *Print()* method should be implemented slightly differently, so the result could be:

```
========================================
Karlo Hansen, Janitor
Monthly salary: 35000
========================================
```

5. Finally, write a class *Director*, representing a director. A director's monthly salary will consist of the agreed monthly salary and a monthly bonus to be passed as an argument to the constructor. The class must therefore have a variable to this value:

```
public class Director extends AbstractEmployee
{
  private int bonus; // monthly bonus
```

The class must also implement the method *Salary*, even though it is implemented in the base class, as it must now operate in a different way. Its value must consist of the value of the base class's method, and the value of the variable *bonus*, and the property can be written as follows:

```
public override int Salary
{
  get { return base.Salary + bonus; }
}
```

You should just accept the syntax, but the word *base* means that you refers to the property *Salary* in the base class.

The *print()* method must be rewritten so that the result for example could be the following, but the value of the monthly salary should be without bonus:

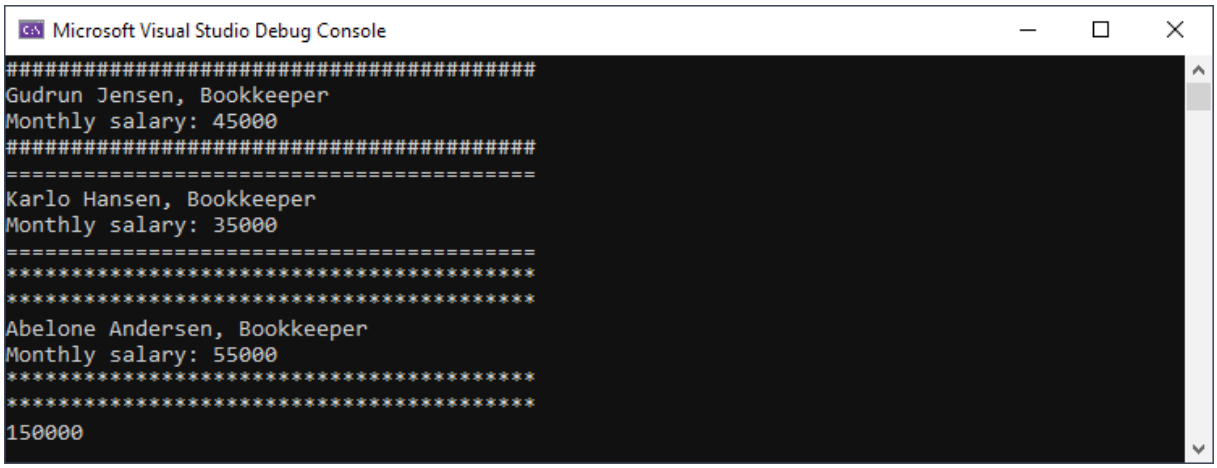
```
****************************************
****************************************
Abelone Andersen, Dircktor
Monthly salary: 55000
****************************************
****************************************
```

6.

Write finally a program (the M*ain()* method) which does the following:

1. Creates an object of the type *Employee*, that is a *Bookkeeper* named *Gudrun Jensen* and with a monthly salary that is 45000
2. Creates an object of the type *Employee*, which is a *Janitor* named *Karlo Hansen* and with a monthly salary that is 35000
3. Creates an object of the type *Employee*, which is a *Director* named *Abelone Andersen* and with a monthly salary that is 55000 and a bonus on 15000
4. Prints the three objects
5. Prints the sum of the three employees monthly salary

If you executes the program, the result should be as shown below:



4.8 EXAMPLE: CUBES

I will show a class representing a usual cube with six sides. The class can be written as follows:

```
using System;

namespace Cubes
{
  class Cube
  {
    private static Random rand = new Random();
    private int eyes;

    public Cube()
    {
      Roll();
    }

    public int Eyes
    {
      get { return eyes; }
    }

    public void Roll()
    {
      eyes = rand.Next(1, 7);
    }

    public override string ToString()
    {
      return "" + eyes;
    }
  }
}
```

The class has two fields: A class variable and an instance variable. The class variable is called *rand* and is an object of the type *Random*. It is a random number generator which can generate random numbers. It works by reading the machine clock and then, produces a sequence of values, based on the clock's value when the object is created. As you can not know (assume) something about what the clock shows at any given time, the start time is randomly and the sequence of generated numbers appear to be random. The variable *rand* must then be a class variable, when objects of type *Cube* otherwise would have their own random number generator, which could easily result in being initialized with the same value of the clock. The result would be that a number of *Cube* objects would create the same sequence of random numbers. The class's instance variable is called *eyes* and is used to keep track of the value of each *Cube*. Since two *Cube* objects do not necessarily have the same value, this variable must be an instance variable, so each object has its own copy.

The class has a method called *roll()*, and simulating that you throw the cube. When the method is performed the class variable *rand* is used. The class *Random* has a method called *Next()*, which returns a random non-negative integer. In this case, it is one of the values 1, 2, 3, 4, 5 and 6, and the result is a random value between 1 and 6 both inclusive.

The class *Cube* also has a constructor. Since the variable *eyes* is not initialized, the default value is 0. This is unfortunate because it is an illegal value. Therefore the constructor rolls the cube and thus ensures that the cube's value from the start is legal.

The class has two other methods. The property *Eyes* is a method that returns what the cube shows. It looks like what you met above. Then there are the method *ToString()*, which returns the value of the cube as a text. The syntax of the return statement is not obvious, but it force the method to return the value of the variable *eyes* as a text. Note that the method is defined with *override*, as it overrides a method from a base class.

The class could be used in a *Main()* method as follows:

```
static void Main(string[] args)
{
  Cube c1 = new Cube();
  Cube c2 = new Cube();
  do
  {
    c1.Roll();
    c2.Roll();
    Console.WriteLine(c1 + " " + c2);
  }
  while (c1.Eyes != c2.Eyes);
}
```

It requires a little explanation when I uses statements which I have not yet mentioned. The first two statements creates two *Cube* objects, and to what has been said above, there is nothing new in it. The next statement is a *do* statement and is an example of a *loop*. It means that the three statements in the block between { and } is performed until the condition after *while* is *false*. The condition uses the operator != meaning different from, and thus the condition is *true*, as long as the two *Cube* objects shows a different number for the eyes. The block with the three statements rolls the cubes and prints them. Here you should note that each iteration of the loop prints

```
c1 + " " + c2
```

and it are the values of the *Cube* class's *ToString()* method that are printed. The result is that the program rolls the two cubes and prints them until they shows the same number of eyes.

A *do* statement is an example of a statement for program control and are discussed in the next chapter, but when you see the code, it is not particularly difficult to find out what happens.

Exercise 6: Coins

In this exercise, you should write a program, similar to the above, but instead of cubes the program should works with coins.

Start with a new Visual Studio project, you can call *Coins*.

Add a class called *Coin* that represent a coin that can show *head* or *tail*, that is a coin that can flip. The class should look likes the class *Cube* and should have a random number generator. In addition, it must have an instance variable to the coin's value:

```
private char value;
```

The value must be *H* or *T*. The class must include a constructor and should have three methods:

- *Value*
- *ToString()*
- *void Roll()*

Here, the first two are written in the same way as in the class *Cube*, but the latter requires a little more, and can be written as follows using the questions operator:

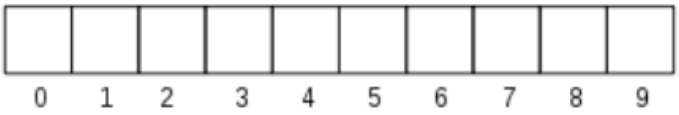
```
public void Roll()
{
  value = rand.Next(2) == 1 ? 'H' : 'T';
}
```

Here, the random number generator is used to return a random number which is 0 or 1, and depending on this value the method assigns the variable *value* the value '*H*' or '*T*'.

You must then write a *Main()* method that creates two *Coin* objects and then uses a *do* loop to throw the coins and print the two coins until they have the same value.

4.9 ARRAYS

An array is a container for objects of a certain type, and an array is itself an object. An array can contains a certain number of elements, and the number of elements an array can contains are defined when the array is created. One may illustrate an array as follows



illustrating an array with room for 10 elements. Each element is identified by an index, starting with 0. Consider as an example the following *main()* method:

```
using System;

namespace Array01
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] arr = new int[10];
      arr[0] = 100;
      arr[1] = 200;
      arr[2] = 300;
      arr[3] = 400;
      arr[4] = 500;
      arr[5] = 600;
      arr[6] = 700;
      arr[7] = 800;
      arr[8] = 900;
      arr[9] = 1000;
      Console.WriteLine(”Element at index 0: ” + arr[0]);
      Console.WriteLine(”Element at index 1: ” + arr[1]);
      Console.WriteLine(”Element at index 2: ” + arr[2]);
      Console.WriteLine(”Element at index 3: ” + arr[3]);
      Console.WriteLine(”Element at index 4: ” + arr[4]);
      Console.WriteLine(”Element at index 5: ” + arr[5]);
      Console.WriteLine(”Element at index 6: ” + arr[6]);
      Console.WriteLine(“Element at index 7: “ + arr[7]);
      Console.WriteLine(“Element at index 8: “ + arr[8]);
      Console.WriteLine(“Element at index 9: “ + arr[9]);
    }
  }
}
```

Here you create an array named *arr*, which has room for 10 elements of the type *int*. After the array is created, all elements has the value 0. The next 10 statements assigns values to all places in the array. If the *Main()* method is performed, you get the result:

The type of an array can be anything, including a class type. As an example could the above program *Cubes* be written as follows:

```
using System;

namespace Array02
{
  class Program
  {
    static void Main(string[] args)
    {
      Cube[] c = new Cube[2];
      c[0] = new Cube();
      c[1] = new Cube();
      do
      {
        c[0].Roll();
        c[1].Roll();
        Console.WriteLine(c[0] + " " + c[1]);
      }
      while (c[0].Eyes != c[1].Eyes);
    }
  }
}
```

where the *Main()* method creates an array with room for two *Cube* objects. When the type of an array is a class type, however, there is an important difference, since the individual elements must be created explicitly. When writing

```
Cube[] c = new Cube[2];
```

Main() creates an array with room for two *Cube* objects, but the two places are empty, which means that they have the value *null*, that indicates that the place is empty (it does not refer to anything). The individual objects has to be created explicitly as shown in the following two statements in the program.

You can also let the compiler create an array from a list of values:

```
int[] arr = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };
Console.WriteLine("Element at index 0: " + arr[0]);
Console.WriteLine("Element at index 1: " + arr[1]);
Console.WriteLine("Element at index 2: " + arr[2]);
```

Here the compiler will create the array with the length that the number of elements in the list indicates and the elements in the array are initialized with the values in the list.

For the array of *Cube* objects, one could write

```
Cube[] c = { new Cube(), new Cube() };
do
{
  c[0].Roll();
  c[1].Roll();
  Console.WriteLine(c[0] + " " + c[1]);
}
while (c[0].Eyes != c[1].Eyes);
```

The following program creates an array with 5 integers of the type *int* and initializes the elements with random numbers, and then the numbers are printed on the screen:

```
using System;

namespace Array03
{
  class Program
  {
    private static Random rand = new Random();

    static void Main(string[] args)
    {
      int[] arr = new int[5];
      for (int i = 0; i < arr.Length; ++i) arr[i] = rand.Next();
      for (int i = 0; i < arr.Length; ++i) Console.WriteLine(arr[i]);
    }
  }
}
```

The program has a random number generator and creates an array with room for 5 elements of the type *int*. Next, the array is filled with random integers. This is done with a *for* loop, which is a control structure that I have not yet mentioned, but it is easy enough to understand how it works. It has three parts. In the first part is defined a variable that is initialized to 0. The second part is a condition that tests whether the value of the variable *i* is less than the number of elements in the array. You must specifically noting how one refers to the number of elements in an array as

```
arr.Length
```

If the condition is true, the statement after the *for* statement is executed, which here is

```
arr[i] = rand.Next();
```

This means that the *i*th element in the array is assigned a value. Next, the third part of the *for* statement is executed and add 1 to the variable *i*, and then the statement test the condition again and everything repeats itself. Since the variable *i* for each iteration is 1 higher, all places in the array are referenced until there are no more. *Main()* has another *for* statement, and it works in principle the same way. It runs through all the array's elements place for place. The first *for* statement initialize the elements, while the other prints them.

As another example, the following program defines an array with three elements of the type *string* and prints them:

```
using System;

namespace Array04
{
  class Program
  {
    static void Main(string[] args)
    {
      String[] arr = { "Svend", "Knud", "Valdemar" };
      for (int i = 0; i < arr.Length; ++i) Console.WriteLine(arr[i]);
    }
  }
}
```

Exercise 7: ArrayDemo

Create a new project in Visual Studio, that you can call *ArrayDemo*.

Add a static method *Test1()* that creates an array with room for 10 elements of the type *double*. The method should then fill the array with random numbers of the type *double*, and finally print the content of the array. Call the method from *Main()* to test it.

Add another method *Test2()*, that from a list creates an array of the type *char* with the first 5 capital letters. The method should then print the array. Test the method from *main()*.

Add a method *Test3()* that create an array with room for 10 elements of the type *bool*. The method must then fill the array with random values and finally print the content of the array.

Write a method that has an array as a parameter

```
static void Print(int[] arr)
{
}
```

The method should print the content of the array, so all elements are on the same line, separated by a space.

Write a last method *Test4()*, that creates an array with room for 10 elements of type *int*. The method must fill the array with random 2-digit integers, and finally the method must print the array by using the above *Print()* method.

4.10 EXAMPLE: CUPPROGRAM

In section 4.8, I have shown a class *Cube* representing a usual cube with six sides. In this section I will show a class that can represent a cup with cubes. It is a container, which may include cubes, and therefore the class internal can be implemented as an array of the type *Cube*. The class must have a method that can simulate that you toss with the cup, and also there must be a method that displays the content of the cup. Correspondingly, the class can be written as follows:

```
namespace CupProgram
{
  class Cup
  {
    private Cube[] arr;

    public Cup(int n)
    {
      arr = new Cube[n];
      for (int i = 0; i < n; ++i) arr[i] = new Cube();
    }

    public void Toss()
    {
      for (int i = 0; i < arr.Length; ++i) arr[i].Roll();
    }

    public bool Yatzy()
    {
      for (int i = 1; i < arr.Length; ++i)
        if (arr[i].Eyes != arr[0].Eyes) return false;
      return true;
    }

    public override string ToString()
    {
      string text = arr[0].ToString();
      for (int i = 1; i < arr.Length; ++i) text += " " + arr[i];
      return text;
    }
  }
}
```

The variable *arr* is an array of the type *Cube*, but it is not created in the declaration but in the constructor. The reason is to make the class more general and the constructor has a parameter that tells how many *Cube* objects the cup must contain. You should note that the constructor explicitly must create the concrete *Cube* objects that happens in a *for* loop. It is necessary, since the statement

```
arr = new Cube[n];
```

simply creates an array with room for *n Cube* objects, but not the concrete *Cube* objects. The method *Toss()* simulates that you toss with the cup. This is done by using a for loop that traverses all the *Cube* objects in the array *arr* and throwing them by calling the method *Roll()* for every *Cube* object. Instead of a method that prints the content of the cup, the class has a *ToString()* method that returns the content of the cup as a *String*, that is the values of the individual cubes separated by spaces.

The class has a method called *Yatzy()* which returns *true* or *false*. The method must test if all cubes are all the same, and it is done by a *for* loop that iterate through the cubes and by means of an *if* statement test whether there is a cube, which is not equal to the first cube (has the same value as the value of the first cube). In the case, all the cubes are not identical, and the method returns *false*. If you iterates throughout the loop, it means that all cubes are equal to the first cube, and thus all cubes are equal, and the method returns *true*.

if is also an example of a control structure, and it will also be explained in the next chapter, but it allows to introduce a condition in a program so that a statement is executed only if the condition is *true*. In this case, the statement is

```
if (arr[i].Eyes != arr[0].Eyes) return false;
```

and after *if* there is a condition in parentheses. If the condition is *true*, this means that the two cubes have a different number of eyes, the next statement is performed, which is a *return* statement. This means that the method ends with the value *false*. If the condition is not *true*, the subsequent statement is simply ignored, and in this case it means that the *for* statement continues.

The method *Yatzy()* is included for the following example that uses the class *Cup*, but really the class ought not have such a method, and verifying that all cubes are the same, should be solved in a different way. The problem is that classes must reflect concepts from the real world, and in practice you can not ask a cup, where all cubes are the same, but you must manually inspect each cube. A method as *Yatzy()* makes the class *Cup* more specialized and it is not as easy to use in other contexts.

As an example of the use of the class *Cup* is shown a program that creates a cup with 5 cubes. Then the program simulates a game, where to toss with the cup until all cubes are the same:

```
using System;

namespace CupProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      Cup cup = new Cup(5);
      do
      {
        cup.Toss();
        Console.WriteLine(cup);
      }
      while (!cup.Yatzy());
    }
  }
}
```

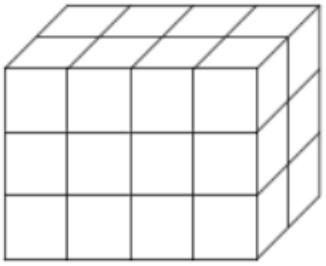
4.11 MULTIDIMENSIONAL ARRAYS

An array as described above is sometimes called a one-dimensional array corresponding to that it is a number or sequence of elements in which the elements are identified by a single index. You can also organize the elements in a table:

2	3	5	7
11	13	17	19
23	29	31	37

and here the individual elements are identified by an index pair consisting of a row index and a column index. For example is the element 17 identified as the element with index (1, 2), that is, an element in row 1 and column 2. One talks in this case of a 2-dimensional array.

You can continue with more dimensions and in principle all the dimensions, as may be needed, and a cube with 3 dimensions is a 3-dimensional array, where the individual elements are identified by a triple (i, j, k):



For arrays of dimension greater than 3 wee have no immediate visual representation, but in programming you can work with arrays of arbitrary high dimensions. In practice, however, it is rare to use arrays with dimensions greater than 2.

In C# you defines a 2-dimensional array as

```
int[,] arr = new int[3, 4];
```

which defines an array with 3 rows and 4 columns. Consider the following method:

```
static void Fill(int[,] arr)
{
  for (int i = 0; i < arr.GetLength(0); ++i)
    for (int j = 0; j < arr[i].GetLength(1);
    ++j) arr[i, j] = rand.Next(10);
}
```

The method has a parameter called *arr*, whose type is *int[,]*, and the methods fills the array with random integers between 0 and 9, both included. You should note how to reference the array elements using an index pair and also how to reference the number of elements in each dimension.

Consider then the following method *Print()*, which prints a two-dimensional array:

```
static void Print(int[,] arr)
{
  for (int i = 0; i < arr.GetLength(0); ++i)
  {
    Console.Write(arr[i, 0]);
    for (int j = 1; j < arr[i].GetLength(1);
    ++j) Console.Write(" " + arr[i, j]);
    Console.WriteLine();
  }
}
```

The two methods *Fill()* and *Print()* are typical examples of using 2-dimensional arrays and the syntax for traversing such an array with two for loops.

You can also create a two-dimensional array from a list:

```
int[,] arr = { { 2, 3, 5, 7 }, { 11, 13,
17, 19 }, { 23, 29, 31, 37 } };
```

which creates an array with three rows and four columns.

In the above examples the 2-dimensional arrays all how rows of the same length, and it is also what you typical needs, but in some situations you can be interested in arrays where it is not the case. A 1-dimensional array has a type, where the type can be anything and here especially an array:

```
int[][] arr = new int[3][];
```

This statement creates an array with 3 elements and where each element is an array. The elements must be created, for example:

```
arr[0] = new int[5]:
arr[1] = new int[4]:
arr[2] = new int[3]:
```

and the result is a 2-dimensional array with 3 rows and the number of columns are 5, 4 and 3.

Exercise 8: Another ArrayDemo

This exercise is a continuation of exercise 7. Make a copy of the solution to exercise 7 and open the copy in Visual Studio.

Implements (that is write the code for) the two methods *Fill()* and *Print()* above. Compile the project so you are sure that there are no syntax errors.

Write a method *Test5()*, which creates a 2-dimensional array with elements of the type *int*, which has 10 rows and 5 columns. Next, fill the array using the method *Fill()* and print it using the method *Print()*. Test *Test5()* from *main()*.

Write a method *Test6()*, which creates a two-dimensional array with four rows and five columns. The array should be created from a list of the first 20 primes. The method must also print the array using the method *Print()*.

Write a method *Test7()*, which creates a 1-dimensional array with elements of type *int[]*, that is elements that are themselves an array. The array must have 5 elements (rows), and the length of the elements (columns) should be 3, 5, 1, 7 and 5. Next, write a new version of the method *Fill()*, which an array of *int* arrays:

```
static void Fill(int[][] arr)
{
  ...
}
```

Note that a class may have several methods with the same name as long they have different parameters. You must also write a method *Print()* which prints such an array. Use these methods in *Test7()* to fill and print the array.

5 PROGRAM CONTROL

In this chapter I will deal with control statements in C#, and in fact I have already used three of them:

- *if*
- *do*
- *for*

The first is used to write conditions in the program, while the last two is for loops. There are two other control statements, and with these statements available, one is able to write programs that perform something concrete and real data processing, and not just programs that prints text on the screen. Therefore, this chapter will largely consist of examples so you are presented for the development of C# applications in practice. Put differently, this chapter focuses on algorithms.

When wee talk about control statements, it is because it's statements, which controls the execution of other statements which partly can be a single statement and also can be a block of statements. A block encapsulates statements using the parentheses { and } and a block of statements can be treated as a whole, for example using control statements. Blocks appear in several places in C#, and for example are the statements in a method a block.

In general statements are performed in the order as they are written in a method, but using control statements, it is possible to deviate from this sequential order, which is necessary in general to be able to write programs that solve something interesting.

5.1 THE IF STATEMENT

C# has 9 control statements, and I have already mentioned and used *if*. As an example you can write something like the following:

```
if (a > b) a -= b;
```

After the word *if* is a condition in parentheses, and the meaning is that the subsequent statement that here is an assignment, only is performed if the condition is *true*. If not, the statement is ignored. As another example, the following code shows an *if* statement that controls a block:

```
if (a > b)
{
  int c = a;
  a = b;
  b = c;
}
```

The block consist of three statements. In general, a block may consist of any number of statements and especially declaration of a variable. In particular, attention should be paid to the fact that if a variable is declared in a block, so it is only known within the block. It is local to the block and is created when the program goes into the block where it is defined. It will be removed again when the program exits the block. Note that the above *if* statement reverses the values of the two variables *a* and *b*, when *a* is greater than *b*.

Formally, the syntax of an *if* statement is

if (condition) block

where *block* is a block of statements. If the block consists of a single statement, it is allowed to omit block boundaries { and } as in the first example above. Some choose because of readability always to use { and }, even if there is only one statement, and it is up to you to choose what you find most readable. The condition must be an expression whose type is *bool*, and thus an expression that evaluates to *false* or *true*.

if-else statement

Another control statement is called an *if-else* statement, but it is little more than an extension of the *if* statement. As an example you can write the following:

```
if (a > b) a -= b; else b -= a;
```

The meaning is that if the condition is *true*, then the statement after the parentheses is executed, and otherwise the statement after *else* is executed. The difference is that *if-else* always does something. Formally, the syntax is:

if (condition) block_1 else block_2

The *if* statement and the *if-else* statement are also called for selections (or branches) corresponding to that they make the execution of statements depending on a condition.

In practice, the *if* statement and the *if-else* statement are the most common control statements, and it is hard to imagine a program where these statements do not occur.

Exercise 9: Sort to elements

Write a program *Sort2* where the user must enter two integers. The program should then print the two numbers in ascending order.

The program should be written as follows:

Create a new project in Visual Studio, as you can call *Sort2*. You must then add a method to the main class to be used for entering a number:

```
private static int Enter()
{
  // create a scanner to entering
  // print a text use the scanner to enter the number
  // convert the value entered to an int and
     return the number - the converting
  // must be done in a try/catch and if there
     is an exception the method should
  // return 0
}
```

You must then write the *main()* method:

```
static void Main(string[] args)
{
  // use the method enter() to enter to numbers
  // if the first number is larger than the other,
     the numbers must be reversed -
  // see above the if how you do that
  // print the two numbers
}
```

When finished, test the program, so you are sure that it does the right thing. The result could, for instance be:

Problem 5: Sort three elements

In this example, you must write a program similar to the above, but where the user instead must enter three numbers and the program must then print the three numbers in ascending order.

That there are three numbers instead of two numbers complicates actually the problem a lot, but you can go as follows.

Create a new project, that you can call Sort3.

You can then take a copy of the input method of exercise 9, but the method should be changed so that in the case of an error (the user has entered an illegal number) the method prints an error message and stops the program.

In *Main()* you can use the method to enter the three numbers *a*, *b* and *c*. You can then sort the three numbers using the following algorithm

```
if a > b then swap the two numbers
if b > c then swap the two numbers
if a > b then swap the two numbers
```

Then the program should print the three numbers *a*, *b* and *c*.

Below is an example of an application of the program where I run it from Visual Studio:

Problem 6: A quadratic equation

In school you learn in mathematics education to solve a quadratic equation, an equation of the form

$$ax^2+bx+c=0$$

where *a* not is 0. In order to solve the equation you has to calculate the discriminant

$$d=b^2-4ac$$

Now that there are no solutions if d < 0, and there is one solution if d = 0, which is determined by the formula

$$x=\frac{-b}{2a}$$

If d > 0, there are two solutions as determined by the following formulas:

$$x_1=\frac{-b-\sqrt{d}}{2a}, x_2=\frac{-b+\sqrt{d}}{2a}$$

The task now is to write a program where the user must enter the coefficients *a*, *b* and *c* of a quadratic equation. If one of the coefficients are not a legal decimal number, the program should stop with an error message. The same should happen if *a* is zero.

Below is an example of running the program:

To solve the exercise you need to know how C# determines the square root of a number. This can be done in the following way:

```
Math.Sqrt(d)
```

where *d* is a variable. You can make the code more simple by changing the input method from problem 5, so you transfer the text as a parameter:

```
private static double Enter(String text)
{
```

Note that the value entered this time should be converted to a *double*, and that the method therefore also need to return a *double*.

5.2 DO AND WHILE STATEMENTS

I have previously used the *do* statement and the syntax is

do
block
while (condition);

The construction executes the block, after which the condition is evaluated. If it is *true* the block is executed again, and it is repeated until the condition becomes *false*. You should note that the block is always executed at least once.

A do statement is an example of an iteration, or loop, which means that a statement or a block is executed several times. Another loop is called a *while* statement, and it has the syntax

while (condition) block

and also means that the block is executed as long as the condition is *true*. An example would be

```
int sum = 0;
int n = 2;
while (n <= 1000)
{
  sum += n;
  n += 2;
}
```

that determines the sum of all positive even numbers which are less than or equal to1000.

Exercise 10: The sum command

You must write a program that can be executed from the command line using the following syntax

```
Sum 999 999
```

where there are two arguments, which are both integers. The program should print the sum of all integers between the two arguments (both included). In the case that there not are two arguments, or the arguments are not legal integers, the program should print an error message.

You can use the following procedure:

Create a new Visual Studio project that you should call *Sum*.

In *Main()*, write an *if-else* statement that tests whether there are two arguments

```
if (args.Length == 2)
{
}
else
{
}
```

In the *else* part the program should print an error message and nothing else. If there are two arguments, they are both strings, and they must be converted to *int* values. It can can raise an exception. This exception should be handled, and for that you need to introduce a *try / catch* block, for the *if* part:

```
if (args.Length == 2)
{
  try
  {
  }
  catch (Exception ex)
  {
  }
}
else
{
}
```

This means that if one of the statements in the try block returns an error, the *try* block is interrupted, and the control is transferred to the *catch* block. In this case, the *catch* block should not do anything except print an error message. The two arguments can be converted into integers as follows:

```
try
{
  int a = int.Parse(args[0]);
  int b = int.Parse(args[1]);
  long sum = 0;
}
```

and note again that in the case of an error (if one of the two arguments is not an integer), then the program jumps to the *catch* block. Note also that when the type of the variable *sum* is defined as a *long*, this is because that the sum of all the integers in a range can quickly become large.

You must finish the *try* block so that you use a *while* statement to determines the sum of all integers from *a* to *b* and print the result.

Problem 7: Enter numbers

You should write a program where the user must enter a number of integers. The entry shall continue until the user enters the number 0. Then the program must print

- how many numbers are entered
- the sum of the numbers
- the smallest number
- the largest number
- and the average of the numbers entered

An example of an execution of the program could be as shown below:

```
Enter a number: 19
Enter a number: 3
Enter a number: 23
Enter a number: 5
Enter a number: 11
Enter a number: 13
Enter a number: 2
Enter a number: 17
Enter a number: 29
Enter a number: 7
Enter a number: 0
Number entered numbers:             10
The sum of entered numbers:         129
The smallest nubmer:                2
The largest number:                 29
The average of the numbers entered: 12.9
```

To enter a number, you can use the input method from problem 6, but it must be changed so that in the case of entering an illegal number it prints an error message, after which the entry must be repeated.

5.3 THE FOR STATEMENT

The last statement to iterations is called *for*, and I have already used the statement many times in connection with arrays. It is the most flexible of the three control statements for loops and also the most used. Formally, the syntax is

for (initialization; condition; expression) block

The initialization is performed only once and will typical initialize one or more variables. The variables created here, are known only in the statement. After the initialization the *for* statement tests a condition, and if it is the *true*, executes the next block. If the condition is not *true*, the statement is ended, and the program continues with the next statement after the *for* statement. After the block have been executed the expression part is performed, and the condition is tested again. In the case where the block consists of only a single statement, it is as for the other control statements allowed not to write the parentheses { and }.

Example: Factorial

As an example is shown a command that has a single parameter. If it is a legal non-negative integer *n*, the command prints the factorial of *n*, where

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
...
```

That is, that *n!* is the product of all positive integers less than or equal to *n*.

```
using System;

namespace Factorial
{
  class Program
  {
    static void Main(string[] args)
    {
      if (args.Length == 1)
      {
        try
        {
          int n = int.Parse(args[0]);
          if (n >= 0) Console.WriteLine(Factorial(n));
          else Console.WriteLine("The argument must be non-negative");
        }
        catch (Exception ex)
        {
          Console.WriteLine("The argument must be an integer");
        }
      }
      else Console.WriteLine("Illegal number of arguments");
    }

    private static long Factorial(int n)
    {
      long t = 1;
      for (; n > 1; --n) t *= n;
      return t;
    }
  }
}
```

Note first that the factorial is calculated by the method *Factorial()*, where the parameter is the number whose factorial to be determined. Otherwise consists the calculation of a *for* loop that calculates the result in the local variable *t*. You should note that the initialization is empty, what is allowed. In fact, all three parts of the *for* statement must be empty. In particular, if the condition part is empty, the condition is always *true*.

Most of the code in the *Main()* method are used to validate the arguments from the command line. First *Main()* tests whether there are the right number of arguments, and if it is not the case, the program stops with an error message. Is there an argument, the program will convert it to an *int*, and is the result a non-negative number, the method *Factorial()* is called to calculate the factorial. In case of an illegal number the program prints an error message.

Example: DigitSum

I want to show another example of the use of a *for* statement which is also a command that is carried out with a single argument. Is this argument a legal non-negative integer, the command calculates the sum of the numbers digits and repeat that calculation until the sum is less than 10:

```
DigitSum(123) = 1 + 2 + 3 = 6
DigitSum(123456) = 1 + 2 + 3 + 4 + 5 + 6 = 21 = 2 + 1 = 3
DigitSum(123456789) = 1 + 2 + 3 + 4 + 5 + 6 + 7 +
                      8 + 9 = 45 = 4 + 5 = 9
```

The *main()* method is in principle the same as above, so I'll just show the method, that calculates the sum of digits:

```
private static int DigitsSum(long t)
{
  for ( ; t > 9; )
  {
    long s = 0;
    for ( ; t > 0; s += t % 10, t /= 10);
    t = s;
  }
  return (int)t;
}
```

This time there are two *for* statements, one inside the other. In the outer *for* statement both the initialization and expression part are empty, and the loop is repeated as long that the variable *t* has more than one digit. The inner *for* statement determines the sum *s* of the digits of the number *t*. Here you should note that

```
t % 10
```

means the last digit, while

```
t /= 10
```

throws the last digit away. Also note that the expression part this time consists of two expressions separated by commas, and the block is empty. As this the statement is probably not the most readable, but legal, it shows a little bit about how flexible the *for* statement is.

for and arrays

Given an array:

```
int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

it is probably the most classic use of a *for* loop to iterate through the array and do something with the individual elements. For example a loop which determines the sum of all elements in the array:

```
int sum = 0;
for (int i = 0; i < arr.Length; ++i) sum += arr[i];
```

It corresponds exactly to what I have shown earlier when I introduced arrays. However, there is a variation of the *for* statement which is written as follows:

```
int sum = 0;
foreach (int t in arr) sum += t;
```

and is a loop doing exactly the same. This variant of the *for* statement works by means of an iterator, but about this later. So far, the only advantage to write the statement in that way is a little shorter notation, and perhaps the statement also is quite more readable and easy to understand. This variant of the *for* statement is called a *foreach* statement.

Exercise 11: Printsum

Write a program where you in the *main()* method defines the following 2-dimensional array:

```
int[,] arr = {
  { 2, 3, 5, 7, 11},
  { 13, 17, 19, 23, 29 },
  { 31, 37, 41, 43, 47 },
  { 53, 59, 61, 67, 71 },
  { 73, 79, 83, 89, 97 } };
```

You must then add a method

```
private static int Sum1(int[,] arr)
{
}
```

that as parameter has a 2-dimensional array. The method should determine and return the sum of the array's elements where the sum must be determined by means of two common *for* loops one inside the other.

Write a similar method

```
private static int Sum2(int[,] arr)
{
}
```

which do the same, but this time the sum must be determined by two *while* statements.

Exercise 12: The Fibonacci numbers

The Fibonacci numbers are a sequence of numbers where the first 11 is shown in the following table:

Index	Value
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

The first two (the 0th and 1st) Fibonacci numbers are defined as respectively 0 and 1. Then the numbers are defined in that way that a certain Fibonacci number is the sum of the previous two.

You must now write a program where the user should enter an integer. The program must then print all Fibonacci numbers whose index is less than or equal to the number entered.

Problem 8: A prime

A prime number is an integer greater than 1, which is only divisible by 1 and itself. To determine whether a number is prime, you can try to divide by any number greater than 1 and less than the number to test for prime and examine whether there is a number that goes up. In this case, it is not a prime number. Going none of the numbers up, it is a prime number.

You can improve the algorithm by noting that 2 is the only even prime, and that it is enough to divide by number less than or equal to the square root of the number.

You should write a program where the user must enter an integer and if it is a legal integer the program must print where the number is a prime. The process should repeat until the user enter 0.

5.4 THE SWITCH STATEMENT

while, *for* and *do* is control structures for loops, and *if* is a control structure for branching or selection. In most cases, *if* is the preferred statement for branching, but there is an alternative called *switch*. It is best explained with an example.

In the following program, the user must enter an integer. If the number is 1, 2, 3, 4, or 5, the program must print the name of the corresponding day, and if the number is 6 or 7, the program must print the text *Weekend*, and otherwise, the program must print an error message.

```
using System;

namespace WeekDay
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.Write("Enter number of the day in the week: ");
      String text = Console.ReadLine();
      switch (text)
      {
        case "1":
          Console.WriteLine("Monday");
          break;
        case "2":
          Console.WriteLine("Tuesday");
          break;
        case "3":
          Console.WriteLine("Wednesday");
          break;
        case "4":
          Console.WriteLine("Thursday");
          break;
        case "5":
          Console.WriteLine("Friday");
          break;
        case "6":
        case "7":
          Console.WriteLine("Weekend");
          break;
        default:
          Console.WriteLine("Illegal day...");
          break;
      }
    }
  }
}
```

When the program starts the user should enter today's number. This is followed by a *switch* statement. After the word *switch* is an expression which here is merely a variable of the type *String*:

```
switch(text)
```

The construction also has a number of *case* entries, where there after each *case* is a constant followed by a colon. The constant must have the same type as the expression (which is here *String*):

```
case "1":
```

When the statement is executed, the expression is evaluated, and the control is transferred to the *case* entry which value is identical to the value of the expression. Hereafter are the next statements executed until one meets a *break* statement or the *switch* statement ends. If there is no *case* entry corresponding to the value of the expression, there are two options. Have the *switch* statement a *default* entry, control is transferred to it. Otherwise the entire *switch* statement is skipped. Note that a *switch* does not need to have a *default* entry. If in the above example 6 or 7 is entered, the program will in both cases prints

```
Weekend
```

If, for example the user enter 6, the program will continue with the next statement after *case 6*, which is *case 7*, as there is no break after *case 6*.

The *switch* statement is a kind of branching with many branches, and it can sometimes be a sensible solution, but not as often as the other control statements. You should note that the type of the expression in *switch* can be all integer types, *char*, *bool*, and *string*.

Exercise 13: Seasons

You should write a program similar to the above, but you have this time to enter the number for a month (1, 2, 3, ..., 12). The program should print the season for the month:

- spring
- summer
- fall
- winter

5.5 RETURN STATEMENT

The *return* statement is already used many times, but it can also be perceived as a control statement. In general the statement interrupts a method and the control is passed to the place where the method was called. Note specifically that a *return* statement in the *Main()* method interrupts *Main()* and thus stops the program. If the method is a *void* method there should not be anything after *return*, and the effect is only that the method is terminated. If, however, the method has a type, then there after the *return* must be an expression of the same type as the method. The method will then return the value of this expression, a value which may then be sent back to the calling code. You should particularly note that a method may well have multiple return statements, typically controlled by an *if* statement.

5.6 BREAK, CONTINUE AND GOTO

In conjunction with the *switch* statement, I have been using *break* as a statement that interrupts a *switch* statement. *break* may also be used to interrupt a loop.

Consider as an example the following method where *rand* is a random number generator, which is created at the start of the program (called *BreakContinue*):

```
private static void Test1()
{
  int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  int elem = rand.Next(1, 30);
  int n = -1;
  for (int i = 0; i < arr.Length; ++i)
    if (arr[i] == elem)
    {
      n = i;
      break;
    }
  if (n >= 0) Console.WriteLine(elem + " found on place number " + n);
  else Console.WriteLine(elem + " not found");
}
```

The method creates an array of 10 integers, and a variable *elem* is initialized with a random number. The next *for* loop checks to see if this number is found in the array, and if so, save the index of the element in a variable *n*. When the element is found, there is no need to search far and I therefore wants to interrupt the loop. This can be done with *break*, and the method continues with the first statement after the loop.

The possibility to break out of a loop in this way is quite reasonable and has many uses. *break* can be used in the same way if the loop is *do* or *while*.

You can also break out of a loop using a *goto* statement. The following method search in an array until it founds a match:

```
private static void Test2()
{
  int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  int n = -1;
  int elem;
  for (; ; )
  {
    elem = rand.Next(1, 30);
    for (int i = 0; i < arr.Length; ++i)
      if (arr[i] == elem)
      {
        n = i;
        goto found;
      }
  }
found:
  Console.WriteLine(elem + " found on place number " + n);
}
```

Note the first *for* loop which is empty. The result is an infinity loop. An iteration starts determining a random number, and the inner *for* loop is used to search the array for this number. If the number is found a *goto* statement is performed, which means the program jumps til label *found*. The result is that the other loop is interrupted.

In general, the use of *goto* is considered as a poor solution, since the statement easily leads to code that is difficult to understand. Therefore, this construction should be used exceptionally only, and the example is included for the sake of completeness only.

Consider then the following method, which just is a helper method in the program:

```
private static string Create()
{
  string text = "abcdefghijklmnopqrstuvwxy1234567890";
  String line = "";
  for (int i = 0, n = rand.Next(10, 40); i < n; ++i)
    line += text[rand.Next(text.Length)];
  return line;
}
```

The method creates a random string whose characters consist of lowercase letters and digits. The string length is between 10 and 39 characters. You must specifically note the method how to reference the individual characters in a string as you do in the same way as for an array. It means that the class *String* overrides the index operator, but more on that later.

```
text[rand.Next(text.Length)]
```

is thus, an arbitrary character in the string

```
"abcdefghijklmnopqrstuvwxy1234567890"
```

There is a control statement that is called *continue* and is in many ways an alternative to *break*. Consider the following method:

```
private static void Test3()
{
  String text = Create();
  String number = "";
  for (int i = 0; i < text.Length; ++i)
  {
    char c = text[i];
    if (c < '0' || c > '9') continue;
    number += c;
  }
  Console.WriteLine(number);
}
```

The method creates a string of random characters using the method *create()*. Then the method iterates through the string in a *for* loop, to build a sub-string consisting of all characters that are digits. When in the loop there is a specific character, that it is not a digit:

```
if (c < '0' || c > '9') continue;
```

you can go to the next character, and here it is done with a *continue*. This means that you transfer the control to the end of the block, and hence skips the statement

```
number += c;
```

continue is not used so often, but it has its uses. Note also that the problem in this case could be solved in other ways than using the *continue*.

Problem 9: A palindrome

A palindrome is a phrase that is spelled the same front and rear, however, so that spaces and special characters are ignored, and that there is no distinction between uppercase and lowercase letters. Examples of palindromes are

- Otto
- Al lets Della call Ed Stella
- Madam, I'm Adam
- A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal-Panama!

You must write a command where the user must enter a line (a text), and the program should test whether this entered text is a palindrome. The program should continue until the user enter the empty line.

Problem 10: Craps

You must write a program that simulates a simple cube game called *Craps*:

A player rolls two cubes, and then the sum of the eyes are noted. If the sum is 7 or 11, the player has won. If the sum is 2, 3 or 12, the player has lost (the house has won). If the sum in contrast is 4, 5, 6, 8, 9 or 10, the sum is recorded as the player's points. The player then proceeds to throw the cubes until the sum of the eyes either are the player's points or until the sum is 7. Is the sum is the player's points, the player won. If the sum is 7, the player has lost, and the house wins.

The program must act in that way that the user first enter how many games he want to play. Then the program simulates the desired number of games, and last prints, how many times the player has won, and how many times the house has won. It must be repeated until the user enter 0 for number of games.

There is however some design requirements for the program since it must consist of several classes. The task must be solved according to the following guidelines:

1. Start by creating a Visual Studio project that you can call *Craps*.
2. Add a class *Cube* to the project, the class must be completely identical to the corresponding class from chapter 3, so you can copy the code from there.
3. Add a class *Cup* to represent a cube cup. The class must be equal the corresponding class from the project *CupProgram*, so you can start by copying the code from this class. The class must then be changed as follows:

   The method *Yatzy()* must be removed.

   Add a property

```
public int Count
```

   that returns the number of cubes in the cup.

   Add a method

```
public Cube GetCube(int n)
```

that returns the *n*th cube in the cup.

4. Add a class *Game*, which represents the game itself, that simulates that the game is played a certain number of times and prints the result.
5. Write the code for the *Main* class where the user can enter the number of times the game should be played.

When finished, it is important that you test the program well. You can not do much else than executes the program many times and evaluate the result.

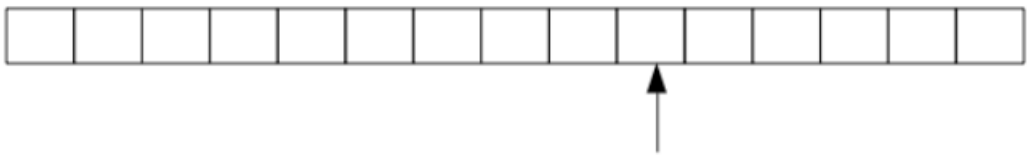
The above version of the game is somewhat simplified, and there are more rules associated with the game. If you play the game, as described above, the player and the house has almost the same probability of winning with a small overweight to the house, and that it should also like to be, if the house not has to go bankrupt.

6 LIST

C# has a family of so-called *collection classes*, which are classes that can be thought of as containers for objects. These classes are discussed later, but I will introduce one of them, as you very often need. The class is called *List,* and indeed it is difficult in practice to write programs without having this class available.

A key concept in C# and, for that matter in any other programming language is an array, which is a data structure that may contain a number of objects of a certain type. The use of arrays is in principle quite straightforward, and there are rarely associated with the major problems to it, but there's one problem, namely that at the time when you create an array, you have to say how many objects it must have room for. It is far from always possible, and in many cases you have to solve the problem by creating arrays that are large enough. In other contexts, it may be necessary to count how many elements an array must have room for, and then create an array of the right size. It can lead to inefficient code, and it is precisely these difficulties that the class *List* can solve.

Technical is a *List* a class which encapsulates an ordinary array, and the class has methods for manipulating the data in that array. You should think of a *List* as illustrated below, where the arrow indicates the next available place:



The basic method is called *Add()*, and it adds an object to where the arrow points and after the object is added the arrow is moved one place right. When you create a new *List* the arrow is at the location to the left, corresponding to that the list is empty. As added objects, move the arrow to the right, and attempts to add an object, and there is no room, the array will automatically expands. That is exactly what is the data structure's strength, and as a programmer you can fill objects into the list without considering whether there is room or not. The class has many other methods, so as an example you can refer to the individual objects in the list with an index and you can by an index insert elements in the middle of the list, delete elements, etc., but it is important to bear in mind that a *List* is a sequence, and that it can not have holes. This means in practice that if you insert an element in the middle of the list, all elements to the right of the new element is moved one place, and in the same way if you delete an element, then all elements to the right of the element that is deleted has to move one place left. In general are the class and its methods

effective, and specifically is *Add()* extremely effective, and as a programmer you should not think about all the technical stuff, but think of the class as a dynamic array. As an example the following method creates a *List* (the program *ListProgram*):

```
public static void Test1()
{
  List<string> list = new List<String>();
  list.Add("Gorm den Gamle");
  list.Add("Harrald Blåtand");
  list.Add("Svend Tveskæg");
  list.Add("Knud den Store");
  list.Insert(3, "Harrald d. 2.");
  Console.WriteLine(list[2]);
  foreach (string s in list) Console.WriteLine(s);
}
```

You should note the syntax of how to create the list, and how to specify that the list should be used for *string* objects. You must specify what it is for a kind of elements that the list should contain. Next the program add 4 elements to the list, but the next statement inserts an element in a particular space where an index indicates where you want to insert the element. The second last statement shows how to refer to an element with an index, and you do that in the same way you do for an array. Finally, the last statement, shows how to iterate through the elements in the list. You must also be aware of the property *Count*, which is not used in this example, but returns the number of elements in the list, and you could for example iterate through the list as follows:

```
for (int i = 0; i < list.Count; ++i) Console.WriteLine(list[i]);
```

As another example the method *Test2()* creates a *List* to integers and initialize the list with 8 integers and then determines the sum:

```
public static void Test2()
{
  List<int> list = new List<int>();
  list.Add(2);
  list.Add(3);
  list.Add(5);
  list.Add(7);
  list.Add(11);
  list.Add(13);
  list.Add(17);
  list.Add(17);
  int s = 0;
  for (int i = 0; i < list.Count; ++i) s += list[i];
  Console.WriteLine(s);
}
```

The class *List* is defined in the namespace *System.Collections.Generic*, and the program must then have a *using* for that namespace.

The class is easy to use, and I will in the following apply it where it is needed and especially I will apply it in the final example.

Problem 11: Standard deviation

If you have a row of numbers $a_1, a_2, \ldots, a_n$, you define the standard deviation as

$$\sigma = \sqrt{\left(\frac{\sum_{i=1}^{n} (a_i - my)}{n} \right)}$$

where *my* is the average value of the numbers:

$$my = \frac{\sum_{i=1}^{n} a_i}{n}$$

Write a program where you can enter any number of decimal numbers (numbers of the type *double*) and store them in a *List*. The entry must continue until you enter 0, after which the program must print the standard deviation of the numbers.

Can you write this program without using a *List*?

7 COMPARISON AND SORTING

If you executes the following method

```
private static void Test01()
{
  String s1 = new String("Knud");
  String s2 = new String("Knud");
  Console.WriteLine(s1);
  Console.WriteLine(s2);
  Console.WriteLine(s1 == s2);
  Console.WriteLine(s1.Equals(s2));
}
```

you get the result:

which is also what you would expect, but you should note that the comparison of the two names occurs in two ways, partly with the comparison operator and partly with the *Equals()* method.

Consider as an example the following class that represents the name of a person:

```
public class Name
{
  private string firstname;
  private string lastname;

  public Name(string firstname, string lastname)
  {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  public string Firstname
  {
    get { return firstname; }
  }

  public string Lastname
  {
    get { return lastname; }
  }

  public override string ToString()
  {
    return firstname + " " + lastname;
  }
}
```

If you executes the following method

```
private static void Test02()
{
  Name n1 = new Name("Olga", "Jensen");
  Name n2 = new Name("Olga", "Jensen");
  Console.WriteLine(n1);
  Console.WriteLine(n2);
  Console.WriteLine(n1 == n2);
  Console.WriteLine(n1.Equals(n2));
}
```

you get the result:

and this is hardly what you would expect since the two *Name* objects represents the same name. If you then add the following method to the class *Name:*

```
public bool Equals(object obj)
{
  if (obj == null) return false;
  if (obj.GetType() == GetType())
  {
    Name name = (Name)obj;
    return firstname.Equals(name.firstname) &&
    lastname.Equals(name.lastname);
  }
  return false;
}
```

and performs the method *Test2()* again you get the result

Now the method *Equals()* returns the expected result. That the method *Equals()* should be written as shown above, you should just accept in this place, but the following happens:

- if the parameter is *null* the two objects can not be *Equals()*
- if the two objects are instances of the same class, the parameter is converted to a *Name*, and the two objects are equal if they has the same first name and the same last name
- else the two objects can not be *Equals()*

When the comparison operator returns *false* it is because the operator compare object references, and *n1* and *n2* are two different objects. The meaning of *Equals()* is that you can override it so that it compares the values of objects instead of their references, and when you writes a class you often has to override *Equals()* en ensure that comparison of objects works correct. When, as shown in *Test1()*, it seems different for strings, it's because the class *String* overrides the equals operator, but it's a whole other story. Note also, when the comparison of *firstname* and *lastname* works, it is because the class *String* implements *Equals()* so that it compares the values.

Consider the following method:

```
private static void Test03()
{
  string[] names = { "Svend", "Knud", "Valdemar", "Harald" };
  foreach (string s in names) Console.Write(s + ", ");
  Console.WriteLine();
  Array.Sort(names);
  foreach (string s in names) Console.Write(s + ", ");
  Console.WriteLine();
}
```

If you executes the method, you get the following result:

This example shows that it is quite simple to sort the array, since the class *Array* has a *Sort()* method. Executing the following method:

```
private static void Test04()
{
  Name[] names = { new Name("Svend", "Grathe"),
  new Name("Knud", "D. 5."),
    new Name("Valdemar", "Den store"), new Name("Harald", "Blåtand") };
  foreach (Name n in names) Console.Write(n + ", ");
  Console.WriteLine();
  Array.Sort(names);
  foreach (Name n in names) Console.Write(n + ", ");
  Console.WriteLine();
}
```

however, it will fail, and the program stops with an exception. The reason is that objects not in general can be compared and be arranged in a sequence. It supports the class *String*, but for your own classes you must define the comparison, and it is also natural, for the system obviously can not know how custom objects should be arranged. If it should be possible to compare objects of the type *Name*, the class must implements an interface:

```
public class Name : IComparable<Name>
{
  private string firstname;
  private string lastname;

  public Name(string firstname, string lastname)
  {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  public string Firstname
  {
    get { return firstname; }
  }

  public string Lastname
  {
    get { return lastname; }
  }

  public override string ToString()
  {
    return firstname + " " + lastname;
  }

  public override bool Equals(object obj)
  {
    if (obj == null) return false;
    if (obj.GetType() == GetType())
    {
      Name name = (Name)obj;
      return firstname.Equals(name.firstname) &&
      lastname.Equals(name.lastname);
    }
    return false;
  }

  public int CompareTo(Name name)
  {
    if (lastname.Equals(name.lastname)) return
    firstname.CompareTo(name.firstname);
    return lastname.CompareTo(name.lastname);
  }
}
```

The class must implement an interface called *IComparable<Name>*, which says that it is possible to compare objects of the type *Name*. The interface defines a single method called *CompareTo()*, which is implemented at the end of the class. The method has a parameter of the type of the objects to be compared, and the method is implemented to meet the following protocol:

1. if the current object is less than the parameter, the method must return a value less than 0 (that is -1)
2. if the current object is greater than the parameter, the method must return a value greater than 0 (that is 1)
3. if the current object is equal to the parameter, the method must return 0

The class *String* implements this interface, and thus the method *Test03()* works. In this case I would compare *Name* objects such that if the two objects have the same last name, it is the first name that determines the order, otherwise the last name.

After the class *Name* is changed to implements *Comparable<Name>* also the method *Test04()* works:

The above examples shows that it is simple to sort an array. The only requirement is that the objects to be sorted, implements the interface *Comparable*. Also objects in a *List* can be sorted, and the following method will sort 4 *Name* objects:

```
private static void Test05()
{
  List<Name> names = new List<Name>();
  names.Add(new Name("Svend", "Grathe"));
  names.Add(new Name("Knud", "D. 5."));
  names.Add(new Name("Valdemar", "Den store"));
  names.Add(new Name("Harald", "Blåtand"));
  foreach (Name n in names) Console.Write(n + ", ");
  Console.WriteLine();
  names.Sort();
  foreach (Name n in names) Console.Write(n + ", ");
  Console.WriteLine();
}
```

You should note that the only difference is that the sorting method *Sort()* this time is a method in the class *List* which sorts the current list. An another important difference for the two sort methods is that the method to sort an array is a static method in the class *Array*, while the method to sort a list is an instance method in the class *List*, but more on that in the book C# 3.

Exercise 14: A sorted cup

Make a copy of the project *CupProgram*. When you run the program, it shows the values of the five cubes in random order. You should now change the program, such that it shows the cubes sorted. You have to change two things:

1. The class *Cube* must implements the interface *Comparable<Cube>*.
2. In the class *Cup* the method *toss()* should sort the cubes after rolled the cubes.

8 FILES

Files are first explained in the book C# 6, but because I use files in the following chapter and also in the following books I will at this place show a little bit about what it takes to use a file, and it is not necessarily much.

Seen from the system a file is a sequence of bytes, and one can in many ways think of a file as an infinitely large array, where the first byte has index 0:

Of course, a file will not be infinite and there is an upper limit determined by the operating system and disk space, but in practice it is limitations that you almost always will ignore. When data is saved in a file and, above all, when data is to be read again, sequences of bytes must be converted from and to the data objects used by the application, and you must specify where in the file to write or read. It's all that kind of things that the C# IO classes take care of, and for many file operations it is simple and goes all by itself, when you just know what it is for data to write.

In the following I will only show how one can write and read text files, and how to write any object to a file and read it again. The last is also known as object serialization / deserialization. In both cases it is simply a matter of accepting the syntax of what to write, but if you can manage text files and serialize objects to a file, you also know most of what is necessary to use the files in your programs.

8.1 TEXT FILES

A text file is a file that contains usual lines of text separated by carriage returns. Seen from C# you use such files by writing and reading one line at a time. If you have to write to a text file, the process is:

1. open the file
2. create the line to be written to the file
3. write the line to the file
4. repeat this two operations as long as there are more lines
5. close the file

As example is shown a method *CreateLine()* which creates a line. The line consists of random integers (between 0 and 99) separated by a semicolon:

```
private static string CreateLine()
{
  string line = "" + rand.Next(100);
  for (int i = 0, n = rand.Next(10); i < n;
  ++i) line += ";" + rand.Next(100);
  return line;
}
```

The number of integers at each line is random, and a line will contain at least one number (and max 10), but otherwise there is nothing to explain.

The next method writes 10 such lines in a file:

```
private static void Test1()
{
  try
  {
    StreamWriter file = new System.IO.StreamWriter("numbers");
    for (int i = 0; i < 10; ++i) file.WriteLine(CreateLine());
    file.Close();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}
```

The method works as described in the above algorithm, and there is not much other than to take note of than this is how to write to a file, but you must be aware of the following:

- *StreamWriter* is a class that represents a text file width the name *numbers*. The first statement opens the file for writing, and in the program the file is called *file* (the variable that references the file).
- You write a line in the file with the method *WriteLine*, and in this case the line written to the file is the line created by the method *CreateLine()* and followed by a *new line*.

- The method *Close()* close the file, and it is important as it may first be at that time that data is physically written to the file.
- The whole is placed in a *try* / *catch*, and it is necessary, when file operations (methods for file processing) typically can raise an exception.

You should note that in principle it is the same thing that will happen every time you have to write lines to a text file. It is only the method *CreateLine()* that will vary.

On the disk the file is called *numbers*, and it is saved in the application's current directory. You should also note that if you executes the program several times, it will every time overwrite an existing file, if there is a file with the same name. If you executes the method, the result could be a file containing the following:

```
49;42;38;69
34
43;85;60;77;91
49;60;16;36;9;3;41;62
72;90;34;50;69;37;18;27;80;90
71;69;42;51;21
47;42;67;35;19;52;29;54;17;74
62;77
87;94;39;94
69;32;2;69;79
```

After creating the file is the next step is to show how to read the content of the file. In principle, the algorithm is:

1. open the file
2. read the next line from the file
3. process the line in the program
4. repeat this two operation as long there are more lines in the file
5. close the file

It is thus almost the same algorithm as when the file is written. In this case, I read all the lines in the file and calculate the sum of the numbers contained in the file.

When you read a line the result is text and the line should therefore be split and each element must be converted to a number. In addition, I have written the following method, and with reference to the algorithm is similar to *process the line in the program*:

```
private static int ParseLine(string line)
{
  int sum = 0;
  string[] elems = line.Split(";");
  foreach (String elem in elems) sum += int.Parse(elem);
  return sum;
}
```

The method is simple enough and contains nothing new, but note, however, a method *Split()*, which is a method defined in the class *String*, that splits a string into sub-strings when the sub-strings are separated by semicolons. The method returns the sum of the numbers in a single line.

The following method reads the content of the file and determines the sum of the numbers:

```
private static void Test2()
{
  try
  {
    int sum = 0;
    StreamReader file = new StreamReader("numbers");
    for (string line = file.ReadLine(); line
    != null; line = file.ReadLine())
      sum += ParseLine(line);
    file.Close();
    Console.WriteLine(sum);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}
```

Also, this method follows the algorithm. This time the file is represented by an object of the type *StreamReader*, which is an object that opens a file for reading. The subsequent *for* loop starts with in the initialization reading the first line of the file. If it goes well, the variable *line* is different from *null*, the line is processed, which is done by calling the method *ParseLine()*. Next, read the next line (the expression part), and it repeats itself until the method *ReadLine()* returns *null*. It happens when you have reached the end of the file. Once all lines are read, closes the file, but in conjunction with reading files it is not quite as important as with writing files, but a good principle.

If the method is performed, the result could be:

```
2825
```

The above example, called *TextFiles*, shows what it takes to write and read a text file, and even though there is a lot more, the above is sufficient to use simple text files in practice.

Exercise 15: A file containing names

You should write a program similar to the above. The program must similarly have two test methods.

The first method must open a file for writing. Next it must perform a loop where the user in each iteration must enter a name and the name should be write to the file. The iteration stops when the user just types enter to the name.

The second method should read the content of the file and prints the names on the screen.

8.2 SERIALIZATION OF OBJECTS

It is also possible to store objects in a file, and actually it is not quite simple. When a file is just a sequence of bytes, this means that the object and its data have to be streamed to a sequence of bytes, a process that is called for *serialization*. It might not be so difficult, but the object must be able to be read again, and it must be possible to convert a sequence of bytes into an object of one type or another. It is only possible if together with the object is saved information about its type and serialization must not only stream the object's data, but also information about the type. With this information it is possible to restore an object from a sequence of bytes, a process that is called *deserialization*. It sounds complicated, and it is that too, but seen from the programmer, it is simple, because C# has the necessary classes, which takes care of it all.

I will write a program that stores a *List* with objects of the type *Name*, where it is the same class that you have seen in chapter 7, however, with one change, where the class now has an attribute which tells that objects of the class can be serialized:

```
namespace ObjectFiles
{
  [Serializable()]
  public class Name : IComparable<Name>
  {
    private string firstname;
    private string lastname;
```

For an object to be serialized, its class must be marked with the attribute *[Serializable()]*. Indeed, it may sound strange, what it should do well, and you must in this book just take note of the need to mark classes in this way if the objects must be able to be serialized.

Consider the following method, which creates a *List* with 4 *Name* objects:

```
private static List<Name> CreateNames()
{
  List<Name> list = new List<Name>();
  list.Add(new Name("Karlo", "Jensen"));
  list.Add(new Name("Gudrun", "Andersen"));
  list.Add(new Name("Valdemar", "Hansen"));
  list.Add(new Name("Olga", "Knudsen"));
  return list;
}
```

The method requires no explanation, but the goal is to serialize the *List* that the method returns. It is possible, because also the class *List* is marked *Serializable*, and because the list contains *Serializable* objects, the entire structure can be serialized as an object. This is done as follows:

```
private static void Test1()
{
  try
  {
    FileStream file = File.Create("names");
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(file, CreateNames());
    file.Close();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}
```

FileStream represents a file, to which you can write objects. The file is created using a static method in the class *File*. To write objects to the file you use a formatter and this time a *BinaryFormatter* (there are other options). The object serialized is the object returned by the method *CreateNames()*, which is a *List* with four *Name* objects. Simpler it can hardly be. You should note that the content of the file this time is not clear text, and if you try to open it, you get a result that you can not easily understand.

As a next step, I will show how to deserialize the object. Consider first the following method that prints the content of a *List* on the screen:

```
private static void PrintNames(List<Name> list)
{
  foreach (Name name in list) Console.WriteLine(name);
}
```

The object can then be deserialized as follows:

```
private static void Test2()
{
  try
  {
    FileStream file = File.OpenRead("names");
    BinaryFormatter bf = new BinaryFormatter();
    List<Name> list = (List<Name>)bf.Deserialize(file);
    file.Close();
    PrintNames(list);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}
```

Again, there is not much to explain, but you can notice that a file where to load objects are represented of an object of the type *FileStream,* but this time opened for reading. The object is read with a *BinaryFormatter*. It returns an *Object*, and it is therefore necessary with a typecast to convert the object to a *List<Name>*, before it is sent to the method *PrintNames()*. You must specifically note the syntax for a typecast, where the type to convert to, is put in parenthesis:

```
List<Name> list = (List<Name>)bf.Deserialize(file);
```

Exercise 16: Write an array to a file

You must write a program similar to the above with a test method to serialize an object and another test method to deserialize the object. Start by writing a method

```
private static int[] CreateNumbers()
{
}
```

that creates and returns an array of the type *int* with 100 numbers when it must be initialized with the values 1, 2, 3, ..., 100.

Then write a method *Test1()* that serialize such an object to a file, that is an object returned from *CreateNumbers()*.

Write a method:

```
private static void PrintSum(int[] arr)
{
}
```

that prints the sum of the numbers in the array on the screen. Write then a method *Test2()* to deserialize the object from before (the array with the 100 numbers) and prints the sum of the numbers on the screen. The result must be 5050.

9 MORE ABOUT CLASSES

C# is an object-oriented programming language where the basic concept is classes and objects. In this book I have mentioned the basic properties of classes, but there is actually much else that you need to know, which is something that the following books will explain. A C# program is a family of classes, and as a programmer you have to write these classes so I will end this book by elaborating on some of the basic concepts, among other things because the next chapter will to a large extent apply classes, and this is explained without all the details.

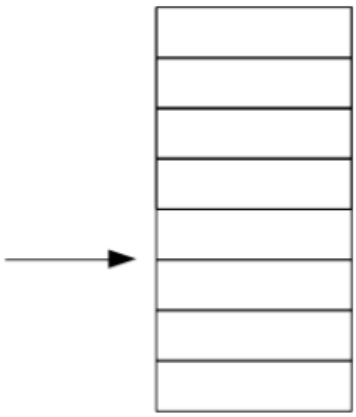
Programs should deal with data, and data must be represented and stored, which happens in variables. A language like C# has a number of built-in types to variables, but often you need to define your own data types that better reflects the task the program must solve. This is where the concept of a *class* comes in. A class is something that define a concept or thing within the program area, and there can be said a lot about what a class is, and what is not, but technically a class is a type. It's a bit more than just a type, as a class partly defines how data should be represented, but also what we can do with the data of that type. A class defines both how the type will be represented, and what operations you can perform on variables of that type.

The chapter will elaborate on the concept of a class through two examples, the first example being continuously expanded through a series of small programs, while the second example is continuous exercises that need solving, but first some details about variables and data.

9.1 THE PROGRAM STACK

When wee creates variables whose type is a class, wee talk about objects, so that a variable of a class type is called an *object*, but really there is no great difference between a variable and an object, and there is well along the road no reason to distinguish between the two. But dig a little further down, there is a reason that has to do with how variables and objects are allocated in the machine's memory.

Variables whose type are *int*, *double*, *bool*, *char*, etc., are called *simple variables* or *primitive types*. To a running program is allocated a so-called *stack* that is a memory area used by the application among other to store variables. The stack is highly efficient, so that it is very fast for the program to continuously create and remove variables as needed. It is called a stack, because one can think of the stack as a data structure illustrated as follows:



When the program creates a new variable, it happens at the top of the stack, where the arrow is pointing, and if a variable must be removed, it is the one that lies at the top of the stack. Variables of simple data types have the property that they always take up the same memory. As an example fills an *int* the same (4 bytes), no matter what value it has. Therefore for that kind of variables are stored directly on the stack as the compiler knows how much room they use. Thus, if you as an example in a method writes

```
int a = 23;
```

then there will be created a variable on the stack of the type *int* and with the room as an *int* requires (4 bytes), and the value is stored there. Variables that in this way are stored directly on the stack, are also called *value types*, and the simple or primitive types, except *String,* are all value types.

Things are different with the variables of a class type. They have to be created explicitly with *new*. If, for example you have a class named *Cube*, and you want to create such an object, you must write

```
Cube c = new Cube();
```

It looks like, how to create a simple variable. The variable is called *c*. When *new* is executed, what happens is that on a so-called heap is created an object of the type *Cube*. You can think of the heap as a memory pool from where to allocate memory as needed. On the stack is created a usual variable of the type *Cube*, but stored on the stack is not the value of a *Cube* object, but rather a reference to the object on the heap. All references take up the same regardless of the type, and they can then be stored on the stack. That is why a

class is called a *reference type*. Where exactly an object is created on the heap is determined by a so-called heap manager that is a program that is constantly running and manages the heap. It is also the heap manager, which automatically removes an object when it is no longer needed.

For the above reasons, it is clear that variables of value types are more effective than the objects of reference types. It in no way means that objects of reference types are ineffective, and in most cases it is not a difference that you need to relate to, but conversely there are also situations where the difference matters. Thus, it is important to know that there are big differences in how value types and reference types are handled by the system, and that in some contexts are of great importance for how the program will behave, but there will be many examples that clarify the difference. So far, it is enough to know that the data can be grouped into two categories depending on their data type, so that the data of value types are allocated on the stack and is usually called variables, while data of reference types are allocated on the heap and called objects, although there is no complete consistency between the two names.

Above I defined the following class, which represents a usual cube:

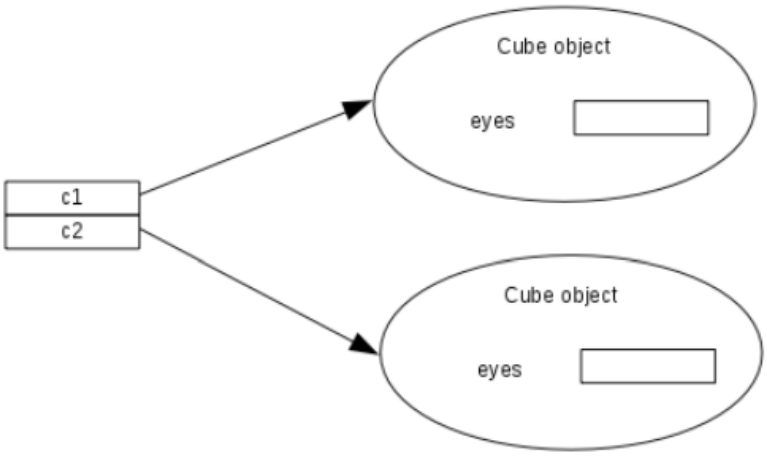
```
class Cube
{
  private static Random rand = new Random();
  private int eyes;

  public Cube()
  {
    roll();
  }

  public int Eyes
  {
    get { return eyes; }
  }

  public void roll()
  {
    eyes = rand.Next(1, 7);
  }

  public override string ToString()
  {
    return "" + eyes;
  }
}
```

The value of the *Cube* or it's state is represented by an *int*, while its functions or behavior is represented by a property and two methods. In a program, you can then create objects of the type *Cube* as follows:

```
Cube c1 = new Cube();
Cube c2 = new Cube();
```

When you create an object of a class, the class's instance variables are created, and then the class's constructor is executed. If a class has no constructor, there automatically will be created a default constructor, a constructor with no parameters. A constructor is characterized in that it is a method which has the same name as the class and do not have any type. A constructor is a method, but it can not be called explicitly and is performed only when an object is instantiated. The class *Cube* has a constructor, that is a default constructor.

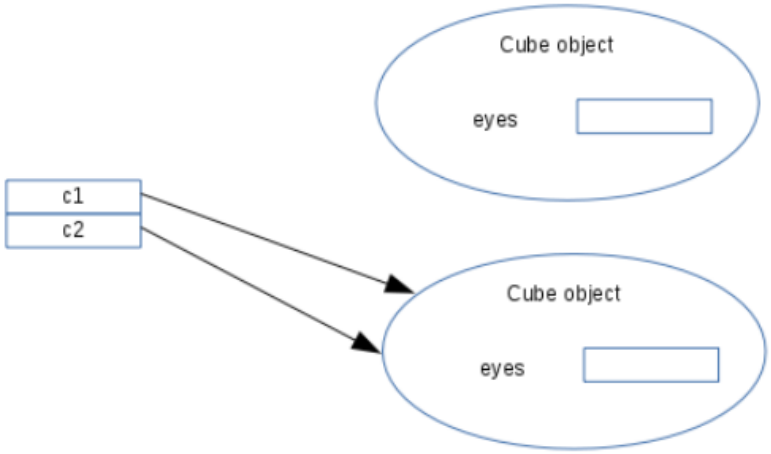
The objects are as mentioned not allocated on the stack, but are created on the heap. *c1* and *c2* are usual variables on the stack, but does not include the objects but include instead references to the two objects on the heap. It's rare you as a programmer need to think about it, but in some situations it is important to know the difference between an object allocated on the stack and on the heap. The figure below illustrate how it looks in the machine's memory with the two variables on the stack that refer to objects on the heap:



If, for example you in the program writes

```
c1 = c2;
```

it means that there no longer is a reference to the object as *c1* before referring to, but there are instead two references to the object as *c2* refers to:



If, you then writes

```
c1.roll();
c2.roll();
```

it means that the same cube is rolled twice because both references refer to the same object.

When there are no longer are any references to an object, it also means that the object is automatically removed from the heap by the heap manager and the memory that the object used, will be unallocated.

A class is referred to as a type, but it is also a concept of design. The class defines objects state as instance variables, how objects are created, and which memory are allocated for the objects. Objects have at a given moment a value as the content of the instance variables, and an object's value is referred to as its state. The class also defines in terms of its methods, what can be done with the object, and thus how to read and modify the object's state. The methods define the object's behavior.

Every C# program consists of a family of classes that together defines how the program would operate and a running program will at a given time consist of a number of objects instantiated from the program's classes, objects that work together to accomplish what the program has to do. The work to write a program is then to write the classes that the program should consist of. Which classes that is, are in turn, not very clearly, and the same program

may typically be written in many ways made up of classes, which are quite different. The work to determine which classes a program should consist of and how these classes should look like in terms of variables and methods is called design. In principle, one can say that if a program does the job properly, it may be irrelevant, which classes it is composed of, but unsuitable classes means

- that it becomes difficult to understand the program and thus to find and fix errors
- that it in the future will be difficult to maintain and modify the program
- that it becomes difficult to reuse the program's classes in other contexts

Therefore, the design and choice of classes, is a very key issue in programming, and in that context wee are speaking of program quality (or lack thereof). A class is more than just a type, but it is a definition or description of a concept from the program's problem area. When you have to define which classes a program must consist of, you must therefore largely focus on classes as a tool to define important concepts more than the classes as a type in a programming language.

An object is characterized by four things:

- a unique identifier that identifies a particular object from all other
- a state that is the object's current value
- a behavior that tells what you can do with the object
- a life cycle from when the object is created and to it again is deleted

An object is created from a class, and is that time assigned a reference that identifies the object. The class's instance variables determine which variables to be created for the object, and the value of these variables is the state of the object. The class's methods define what you can do with the object and thus the object's behavior. The last point of concerning is objects life cycle, which is explained later.

To program in that way in which you see a program composed of a number of cooperating objects, is called *object-oriented programming*. For many years wee have used object-oriented programming, and although it is primarily a question of design, it also requires that the current programming language supports the object-oriented thinking, and this is what makes C# an object oriented programming language.

9.2 CLASS EXAMPLES

I'll start with a class that can represent a course at an educational institution:

```
using System;

namespace Students1
{
  /*
   * Class which represents a subject associated with an education.
   * A subject consists in this context of an id, which is an acronym
   * that identifies the subject, and a name.
   */
  public class Subject
  {
    private string id; // the subject id
    private string name; // the subject's name

    /*
     * Creates a new subject. A subject must have
       both an ID and a name, and is it
     * not the case, the constructor raises an exception.
     *     id The subjects id
     *     name The subjects name
     * Throws an Exception if the id or the name are illegal
     */
    public Subject(string id, string name)
    {
      if (!SubjectOk(id, name))
        throw new Exception("The subject must
        have both an ID and a name");
      this.id = id;
      this.name = name;
    }

    /*
     * Returns the subject's id
     */
    public string Id
    {
      get { return id; }
    }

    /**
     * Returns the subject's name
```

```
     */
    public string Name
    {
      get { return name; }
      set
      {
        if (!SubjectOk(id, name))
          throw new Exception("The subject must have a name");
        name = value;
      }
    }

    /**
     * Represents this object as a string.
     */
    public override string ToString()
    {
      return name;
    }

    /*
     * Method that tests whether two strings can represent a subject.
     *    id The subjects id
     *    name The subjects name
     * Return true, if id and name represents a legal subject
     */
    public static bool SubjectOk(string id, string name)
    {
      return id != null && id.Length > 0 &&
      name != null && name.Length > 0;
    }
  }
}
```

The class is member of a project called *Students1*. Compared to what I have said about classes above there is not much new to explain, and the meaning of variables and methods are explained in the class as comments, but you must be aware of the following.

Classes are defined by the reserved word *class*, and a class has a name as here is *Subject*. In C#, it is recommended that you write the name of classes capitalized, but otherwise applies the same rules for classes names as for the names of variables. A class can have a visibility that may be *public* or *private*, but so far I will define all classes *public*. Visibility says something about who may use the class.

Classes consist of variables and methods, and there is in principle no upper limit for any of the two parts. Both variables and methods can have *public* and *private* visibility. If a member (variable or method) is *public*, it can be referenced from methods in all other classes, that have an object of the class, and if it is *private*, it can only be referenced by methods in the class itself. Usually you define variables as *private*, while the methods to be used by other classes, are defined *public*. Put a little different a class defines with *public* methods, what can be done with objects of the class and thus the objects behavior.

In this case, the class has two variables, both of which are of the type *string* (*string* or *String* is itself a class). Such variables are called *instance variables*, and each object of the class has its own copies of these variables whose values are the object's *state*. The variables do not necessarily refer to anything and such variables has the value *null* (*null* reference), which simply means that you have reference variables, which does not refer to an object.

A class has methods, but can also have properties, which is nothing but another syntax for simple methods. As an example is *Id* a property:

```
public string Id
{
  get { return id; }
}
```

A property can have a *get* part and a *set* part. In this case there is only a *get* part, which means that others can read the value of the *private* variable *id*, but not change the variable. Another property is *Name*:

```
public string Name
{
  get { return name; }
  set
  {
    if (!SubjectOk(id, name)) throw new Exception("The
    subject must have a name");
    name = value;
  }
}
```

This property has both a *get* part and a *set* part so others both can read and change the variable *name*. As mentioned, properties are nothing but simple methods and in principle they can be used for anything, but basically they are linked to instance variables so that others can read and possibly change these variables. Variable names starts with lowercase letter, and if you write properties to a variable, you use the convention that the property name is the same as the variable's name, but starting with the same uppercase letter.

Classes can have one or more constructors that are special methods executed when creating an object of the class. A constructor is written in the same way as other methods, but the name is the same as the class, and they do not have any type. Constructors can in principle perform anything, but they are typically used to initialize instance variables in the class. In this case, the constructor has parameters for an object's values. These parameters have the same names as the instance variables (it is not a requirement, and the parameters could be named anything), and in the constructor's code there are two things with the same name (both an instance variable and a parameter). It is solved by the word *this* where, for example *this.name* refers to the instance variable, while *name* refers to the parameter. I will insist that a specific subject must have both an id and a name. Therefore, the constructor tests the parameters, and if they not both how a value the constructor raises an exception. Exceptions are discussed later. If the parameters do not have legal values, the constructor raises an exception

```
if (!SubjectOk(id, name))
  throw new Exception("The subject must have both an ID and a name");
```

You should note that the parameters are tested by the method *SubjectOk()*. This method is defined *static*, meaning that it can be used without having created an object of the class. Other classes can use the method to test values before creating the object. That kind of control methods may be reasonable, if you later find out that the controls should be different. You should then simply change the method, and it is not necessary to change elsewhere in the code.

That a constructor in this way raises an exception is a guarantee that no one instantiates illegal objects, the one that instantiates an object, must necessarily decide what should happen in case the object is illegal. However, you can discuss the wisdom of in this way to let constructor validate input parameters and possible raise an exception, and it is a discussion I will return to later.

In addition to the constructor and the two properties and then the above static method, the class has only one method *ToString()*. This method is also a bit special as it returns a representation of a specific object as a string and you then have the possibility to print an object. A class do not need to have a *ToString()* method, but many class's have. You should note that the method is defined *override* which means that the method overrides a version of the method from a base class.

Objects must be created or instantiated, which is done with *new*:

```
using System;

namespace Students1
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        Subject subject = new Subject("MAT7", "Mathematichs");
        Console.WriteLine(subject);
        subject.Name = "Matematichs 7";
        Console.WriteLine(subject);
      }
      catch (Exception ex)
      {
        Console.WriteLine("Error: Illigal object");
      }
    }
  }
}
```

The program creates an object *subject* of the type *Subject*. When the constructor in the class *Subject* may raise an exception, you should handled the exception and place the code that may raise an exception in a *try / catch*. After the object is created, it is printed on the screen. If you prints an object with *WriteLine()*, it is the value of the object's *ToString()* method that is printed. In principle, all classes has a *ToString()* method which returns a text that represents a particular object. Finally the program changes the name of the subject, and the object is printed again.

Exercise 17: A Publisher

Create a new project in Visual Studio that you can call *Library1*. You must add a class that should be called *Publisher* that represents a book publisher when the publisher must have the following two properties:

1. An integer, that you can call *code* which is to be interpreted as a publisher identifier. It must be a read-only property and then at property with only a *get*.
2. A name, that just is a text and is the publishers name. It must be a read-write property and as so a property with both *get* and *set*.

It is a requirement that the publisher number is a positive integer and a publisher must have a name. The class must have a constructor that has parameters for both variables and a *ToString()* method that returns the publishers name followed by the publisher number in square brackets

When you have written the class, you should write a *Main()* method that

- Creates a *Publisher* object, you decides the values
- Prints the publisher
- Change the publishers name
- Prints the publisher again

If you executes the program, the result could be the following:

```
The new Publisher [123]
The old Publisher [123]
```

9.3 MORE CLASSES

I will then look at a class representing a subject that a student has completed or participates in. It is assumed for simplicity that the same subjects only can be held once a year and therefore are identified the subject's ID and the year. The project is called *Students2* and contains the class *Subject* as described above. There is added the class *Course* as shown below, and again explains the comments of the individual methods there purposes:

```
using System;

namespace Students2
{
  /**
   * Class that represents a course, where a course
     regarding a year and a subject.
   * A student may have completed or attend a particular course.
   * It is an assumption that the same subjects
     only be executed once a year.
   * A course can also be associated with a
     score. If so, it means that the
   * student has completed the course.
   */
  class Course
  {
    private int year;          // year for when the course
    private Subject subject;   // the subject that the course deals
    private int score =
    int.MinValue;              // the score obtained in the subject

    /*
     * Creates a course for a concrete subjects
       and a given year. The constructor
     * raises an exception if the year is illegal or the subject is null
     *    year The year for the course
     *    subject The subject for this course
     */
    public Course(int year, Subject subject)
    {
      if (!CourseOk(year, subject)) throw new Exception("Illegal course");
      this.year = year;
      this.subject = subject;
    }

    /*
     * Creates a course for a concrete subjects
       and a given year. The constructor
     * raises an exception if the year is illegal
       or the parameters for the
```

```
     * subject is illegal.
     *    year The year for the course
     *    id The subject's id
     *    name The subject's name
     */
    public Course(int year, String id, String name)
    {
      subject = new Subject(id, name);
      if (!CourseOk(year, subject)) throw new Exception("Illegal year");
      this.year = year;
    }

    /*
     * A cource is identified by the subjects id and the year separated by
     * a hyphen.
     */
    public string Id
    {
      get { return year + "-" + subject.Id; }
    }

    public int Year
    {
      get { return year; }
    }

    /*
     * Return true, if the student has completed the course
     */
    public bool HasCompleted
    {
      get { return score > int.MinValue; }
    }

    public int Score
    {
      get
      {
        if (score == int.MinValue)
          throw new Exception("The student has not completed the course");
        return score;
```

```
      }
      set
      {
        if (!ScoreOk(value)) throw new Exception("Illegal score");
        score = value;
      }
    }

    public void SetScore(string score)
    {
      try
      {
        int number = int.Parse(score);
        if (!ScoreOk(number)) throw new Exception("Illegal score");
        this.score = number;
      }
      catch (Exception ex)
      {
        throw new Exception("Illegal score");
      }
    }

    public override string ToString()
    {
      return subject.ToString();
    }

  /**
   * Tests whether the parameters for a course are legal
   * @param year The year for the course
   * @param subject The subject that this course deals
   * @return true, If the year is illegal or the subject is null
   */
    public static bool CourseOk(int year, Subject subject)
    {
      return year >= 2000 && year < 2100 && subject != null;
    }
```

```
    // Validates whether a score is legal
    private bool ScoreOk(int score)
    {
      return true;
    }
  }
}
```

The class looks like the class *Subject*, but there are still things that you should note. The class has three instance variables, and the one of the type *Subject*. You should therefore note that the type of an instance variable may well be a user-defined type and here a class. There is also an instance variable called *score* that has the type *int*. It should be applied to the score which a student has achieved in the subject. In one way or another it should be possible to indicate that a student has not yet been given a score, for example because the course has not been completed. This is solved by assigning the variable *score* the value *int. MinValue* which is the least 32-bit integer that in C# can occur as an *int*. This number is --147483648, and one must assume that this score can not occur in a score system.

Note that wee in C# use the same rules and conventions for method names that wee use for variables but that such a name of a method should start with an uppercase letter.

When you assign a course a score it is validated by the method *ScoreOk()*. This method is trivial, since it always returns *true*, and it provides little sense. The goal is that the method at a later time should be changed to perform something interesting, and you can say that the problem is that the class *Course* not have knowledge of what are legal scores. There are several score systems.

Now I have defined two classes regarding students, and there is a connection between the two classes so that the clas s *Course* consist of an object of the class *Subject*. It is sometimes illustrated as

If you have to create a *Course*, you must provide a year and a *Subject*, which is an object that must also be created. If you find it appropriate, you can add an additional constructor that creates this object, what is the case above corresponding to the class having two constructors. A constructor can be overloaded which means the constructors must have different parameters.

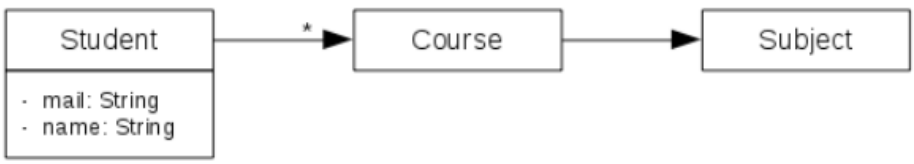
In general a method may be overloaded. In C#, a method is identified by its name and its parameters. There may well in the same class be two methods with the same name as long as they have different parameters, that is when the parameter lists can be separated either on the parameters types or the number of parameters.

Below is a method that creates two courses:

```
private static void Test1()
{
  try
  {
    Course course1 = new Course(2015, new
    Subject("MAT6", "Matematik 6"));
    Course course2 = new Course(2015, "MAT7", "Matematik 7");
    course1.Score = 7;
    Print(course1);
    Print(course2);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}

private static void Print(Course course)
{
  Console.WriteLine(course);
  if (course.HasCompleted)
    Console.WriteLine("The course is completed
    with the result " + course.Score);
}
```

Note how to create objects of the type *Course* and how the two different constructors are used. You should note that the property *Score* may raise an exception and therefore, the *Print()* method may raise an exception, and if it happens the method will be interrupted and the *Exception* object is passed on to the method *Test1()*. Also note how in the *Print()* method you refers to methods in the class *Course* using the dot operator on the *course* object, and that the methods works on the specific *Course* object's state.

I will then look at a class that represents a student. A student must in this example alone have a mail address and a name, and also a student could have a number of courses that the student has either completed or participates in. This can be illustrated as follows:



where * indicates that a student may be associated with many *Course* objects. The class is shown below, and it takes up a lot, particular because of the comments, and they are not shown here, and you are encouraged to investigate the finished class, but I think the meaning of the individual methods is mostly self explanatory

```
using System;
using System.Collections.Generic;

namespace Students2
{
  class Student
  {
    private static int nummer = 0;              // the last used id
    private int id;                             // the students id
    private String mail;                        // the mailadresse
    private String name;                        // the student's name
    private List<Course> courses = new
    List<Course>();                             // a list with courses

    public Student(String mail, String name, params Course[] course)
    {
      if (!StudentOk(mail, name)) throw new Exception(" ... ");
      this.mail = mail;
      this.name = name;
      foreach (Course c in course) courses.Add(c);
      id = ++nummer;
    }

    static Student()
    {
      nummer = 1000;
    }

    public int Id
    {
      get { return id; }
    }

    public string Mail
    {
      get { return mail; }
      set
      {
        if (value == null || value.Length == 0)
        throw new Exception(" ... ");
        mail = value;
      }
```

```
}

public string Name
{
  get { return name; }
  set
  {
    if (value == null || value.Length == 0)
    throw new Exception(" ... ");
    name = value;
  }
}

public int Count
{
  get { return courses.Count; }
}

public Course this[int n]
{
  get { return courses[n]; }
}

public Course this[string id]
{
  get
  {
    foreach (Course c in courses) if (c.Id.Equals(id)) return c;
    throw new Exception("Course not found");
  }
}

public List<Course> GetSourses(int year)
{
  List<Course> list = new List<Course>();
  foreach (Course c in courses) if (c.Year == year) list.Add(c);
  return list;
}

public void Add(Course course)
{
  foreach (Course c in courses) if (course.Id.Equals(c.Id))
```

```
        throw new Exception("The course is already added");
      courses.Add(course);
    }

    public override string ToString()
    {
      return "[" + id + "] " + name;
    }

    public static bool StudentOk(string mail, string name)
    {
      return mail != null && mail.Length > 0 &&
      name != null && name.Length > 0;
    }
  }
}
```

The class *Student* represents a student. The class has four instance variables, where the first is an identifier (a number), the next two are of the type *string* and is respectively the mail address and the student's name. The last is of the type *List<Course>*, and should be used to the courses that the student has completed or participates in. You should note the last parameter to the constructor:

```
params Course[] course
```

The syntax means that the constructor can have a variable number of *Course* parameters, and in the method these parameters are referenced with the same syntax as for an array.

An object has a variable *id*, which is a number that can identify a student. This number is automatically assigned in the constructor starting with 1001 and such that the next student is assigned the next number. How it exactly works, I will return to in connection with static members.

The class has properties for the variables *id*, *mail* and *name* and for these properties there is nothing new to note. The class also has a property *Count* which is the number of courses for the current student. A class can have an indexed property, and the syntax is not entirely obvious:

```
public Course this[int n]
{
  get { return courses[n]; }
}
```

The property is read-only and returns the *n*th course. It means that the class *Student* can be used with the same syntax as it was an array. When a class defines an indexed property it is sometimes called an indexer. An indexer can be overloaded as other methods and here it is overloaded for a *string* which is interpreted as the course ID.

The class has two additional methods in which the first returns a student's courses for a particular year, while the second is used to add a course to a student.

Below is a test method, which creates two students:

```
private static void Test2()
{
  try
  {
    Student stud1 = new Student("svend@mail.dk", "Svend Andersen");
    Student stud2 = new Student("gorm@mail.dk", "Gorm Madsen",
      new Course(2015, new Subject("PRG", "Programming")),
      new Course(2015, new Subject("OPS", "Operating systems")));
    stud1.Add(new Course(2014, new Subject("DBS", "Database Systems")));
    stud2.Add(new Course(2014, new Subject("WEB", "Web applicattions")));
    stud1["2014-DBS"].Score = 4;
    stud2["2014-WEB"].Score = 10;
    stud2["2015-OPS"].Score = 2;
    Print(stud1);
    Print(stud2);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}

private static void Print(Student stud)
{
  Console.WriteLine(stud);
  for (int i = 0; i < stud.Count; ++i)
  {
    Course c = stud[i];
    Console.WriteLine(c + ", " + (c.HasCompleted ? "Score: " +
      c.Score : "Not completed"));
  }
}
```

The first student is created without specifying courses. The other student is created with two courses. Next, the program adds a course to both students and finally assigns scores for three courses, and the two students are printed. If the method is executed, the result is:

```
[1001] Svend Andersen
Database Systems, Score: 4
[1002] Gorm Madsen
Programming, Not completed
Operating systems, Score: 2
Web applicattions, Score: 10
```

Exercise 18: An Author

This exercise is a continuation of exercise 17. Start by making a new project *Library2* and add the class *Publisher* from exercise 17 to this project. You must add a class named *Author*. The class should have three instance variables

1. *id* which is an integer, that can identify an author
2. *firstname* for the author's first name
3. *lastname* for the author's last name

It is a requirement that the first name is not *null*, but it is allowed to be blank. On the other hand, an author must have a last name.

The class should have a constructor, which has two parameters, respectively to initialize *firstname* and *lastname*. An author's ID should be assigned automatically, so that every time a new *Author* is created the object get an id, which is one greater than the previous one. You can solve this using a *static* variable:

```
private static int counter = 0;
```

which is counted up by 1 each time a new *Author* is created.

Regarding the methods, the class must have a read-only property for *id* and read-write properties for *firstname* and the *lastname*. Finally, there must be a *ToString()* method that returns an author's first and last name separated by a space.

Once you have written the class, you must document it with comments.

In the main class, write a method that can print an *Author*:

```
private static void Print(Author a)
```

when the method should print the author's id listed in brackets, as well as the author's first and last name.

Add a class *Book* that represents a book with the following characteristics:

- Isbn, that must be declared
- Title, that must be declared
- Release year, that must be 0 or lie between 1900 and 2100, 0 indicates that release year is unknown
- Edition, which must be between 0 and 15 (inclusive), 0 indicates that the edition is unknown
- Number of pages that must be non-negative, 0 indicates that number of pages are unknown
- Publisher, there must be a *Publisher* object or *null*, where *null* means that the publisher is unknown
- A number of authors, which must be *Author* objects (the number of authors must be 0 if no authors are known)

The class must have four constructors with the following signatures:

```
public Book(String isbn, String title)
{
  ...
}

public Book(String isbn, String title, int released)
{
  ...
}

public Book(String isbn, String title, Publisher publisher)
{
  ...
}

public Book(String isbn, String title, int
released, int edition, int pages,
  Publisher publisher)
{
  ...
}
```

There must be a read-only property to *isbn* and read-write properties for the fields *title*, *released*, *edition*, *pages* and *publisher*.

For authors the class should in the same manner as the class *Student* have a *List<Author>* to *Author* objects:

```
private List<Author> authors = new List();
```

and the class must have a read-only property for this list.

The class *Book* should have a *ToString()* method which returns the book's ISBN and the *title* separated by a space.

In the main class, write a method that can create and return a *Book* object from information on the book's properties:

```
private static Book Create(string isbn, string
title, int released, int edition,
  int pages, Publisher publisher, params Author[] authors)
{
}
```

You must then write a method that can print a book:

```
private static void Print(Book book) {}
```

when it has to print

- the book's ISBN
- the book's title
- the book's release year if known
- the book's edition if known
- the book's pages if known
- the book's publisher if known
- the book's authors if there are authors

You must finally write the *Main()* method, so it performs the following:

- create a book using the above method where you specify values for all the book's fields and including the book's author (s)
- create a book where you only initialize the book's ISBN and title
- prints the two books
- change the release year, edition, number of pages and the publisher for the last book and adds the author (s)
- prints the last book again

The result could, for instance be as shown below:

```
ISBN:       978-1-59059-855-9
Title:      Beginning Fedora From Noice to Professional
Released:   2007
Edition:    1
Pages       519
Publisher:  The new Publisher [123]
Authors:
[1] Shashank Sharma
[2] Keir Thomas

ISBN:       978-87-400-1676-5
Title:      Spansk Vin

ISBN:       978-87-400-1676-5
Title:      Spansk Vin
Released:   2014
Edition:    1
Pages       335
Publisher:  Politikkens Forlag [200]
Authors:
[3] Thomas Rydberg
```

Problem 12: Validate ISBN

A book is characterized by an ISBN, which is an international numbering on 10 or 13 digits. The system was introduced around 1970, and until 2007 it was on 10 digits, which was divided into four groups separated by hyphens:

```
99-999-9999-9
```

wherein the last group always consists of a single digit (character) and is a control character. The other three groups are interpreted as follows:

1. the first group is a country code
2. the second group is the publisher identifier
3. the third group is the title number

The control character is calculated by the modulus 11 method, using the weights 10, 9, 8, 7, 6, 5, 4, 3, 2 and 1. This is best explained with an example. Consider a concrete ISBN:

```
1-59059-855-5
```

To determine the check character you determine the following weighted sum:

```
 1     5     9     0     5     9     8     5     5
10     9     8     7     6     5     4     3     2
-------------------------------------------------------
10  +  45  +  72  +  0  +  30  +  45  +  32  +  15  +  10  =  259
```

You now determines the rest by dividing by 11 and you get

```
259 % 11 = 6
```

If this rest is 0, the check digit is 0. If the remainder is 10, X is the check character, and otherwise, one uses 11 minus the remainder as check digit. As the rest this time is 6, you get the check digit 11-6 = 5.

Eventually it was realized that by the years there would be too few ISBN numbers, and therefore it was decided to change the system from 2007. From that time the numbers are preceded by a 3-digit prefix, and the numbers thus came to consist of another group, that as always has 3 digits:

```
999-99-999-9999-9
```

So far only the numbers 978 and 979 are used as prefixes, but in the future there will probably be other opportunities. In addition to increasing the numbers with this prefix the calculation of the control character was also changed, so that as weights using alternating values of 1 and 3, and the control character is then the modulus 10 of the weighted sum. Again, it is easiest to illustrated the calculations with an example. Consider

```
978-1-59059-855-9
```

You calculate the following weighted sum:

```
 9     7     8     1     5     9     0     5     9     8     5     5
 1     3     1     3     1     3     1     3     1     3     1     3
-----------------------------------------------------------------------
 9  +  21  +  8  +  3  +  5  +  27  +  0  +  15  +  9  +  24  +  5  +  15  =  141
```

Then you calculate rest at division by 10:

```
141 % 10 = 1
```

If this rest is 0, it is control character. Or is the check digit 10 minus the remainder, and in this case, it is 10-1 = 9.

You must now create a new project *Library* and add the classes *Publisher*, *Author* and *Book* from the above exercise to the new project. The class *Book* has a static method *IsbnOk()* to validate whether a string is a legal ISBN. The control is trivial, but you must now change the code to validate an ISBN following the above rules. Implementing the control is not very simple and you will probably be missing methods regarding the *String* class that I have not mentioned. If necessary, seek help in the online documentation.

You must also write a test method that tests whether the control method validates correctly. It is important that you test with multiple numbers, so you get all the cases, and it is important that you also test illegal numbers.

9.4 METHODS

In the explanation of classes I have already dealt with methods, but there are a few concepts that you should be aware of. As mentioned, a method is identified by its name and the parameter list. The parameters that you specify in the method definition, are called the *formal parameters* and they defines the values to be transferred to a method. The values that you transfer when the method is called, is referred to as the *actual parameters*. Above

I have shown how to specify that a method has a variable number of parameters, which is really just a question that the compiler creates an array as the actual parameter. Methods parameters can generally be of any type, but you should be aware that primitive types and class types are treated differently. For primitive types the transmitted values are directly copied, and that is, that on the stack are created a copy of the parameters and the values of the actual parameters are copied to these copies. This means that if a method is changing the value of a parameter that has a primitive type, then it is the value of the copy on the stack that is changed, and after the method is terminated, then the values of the calling code are unchanged. Wee therefore also call a parameter of a primitive type for a value parameter. What a primitive type really is is explained later in book C# 3.

Class parameters are in principle transferred in the same way, where there on the stack is created a copy of the parameter and the current value is copied to this. However, you should be aware of what is being created and copied. In the case of a reference parameter what is created on the stack is a reference, and what is copied is the reference to the current object.

Although parameters of primitive type and classes are treated slightly differently, the parameter transfer in C# is basic value transfers, which means that if a method change the value of a parameter, the change is unknown outside the method. A good example is a swap method, and thus a method that must reverses the two values. Consider the following method

```
public static void Swap(int a, int b)
{
  int t = a;
  a = b;
  b = t;
}
```

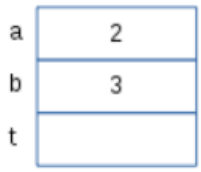
which has two parameters of the type int. If the method is executed as follows:

```
static void Main(string[] args)
{
  int t1 = 2;
  int t2 = 3;
  Swap(t1, t2);
  Console.WriteLine(t1 + " " + t2);
}
```

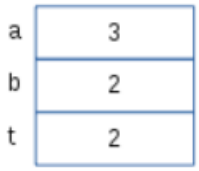
the last statement prints

```
2 3
```

and the two numbers are not reversed. When the method *Swap()* is called, the variables *t1* and *t2* are transferred as actual parameters. The method *Swap()* has two parameters and a local variable, and has therefore three fields that are allocated on the stack, and the values of the actual parameters are copied to the stack:



The *Swap()* method works in that it copies the value of the parameter *a* to the local variable *t*, and then copy the value of *b* to *a*. Finally the value of *t* is copied to *b*, and then the content of the stack is as shown below.



This means that the two values are reversed, but it happened on the stack, and after the method is terminated the three elements are removed from the stack, and the changes are lost. This means that the variables in the calling code is unchanged.

However, C# supports reference parameters where the changes a method makes to the parameters will affect the actual parameters. The method *Swap()* could be written as:

```
public static void Swap(ref int a, ref int b)
{
  int t = a;
  a = b;
  b = t;
}
```

and the only difference is that each parameter is preceded by the word *ref*. If you then executes the method

```
Swap(ref t1, ref t2);
Console.WriteLine(t1 + " " + t2);
```

you will see that the values of the two variables *t1* and *t2* are exchanged. You must note that also the actual parameters must be preceded by the word *ref*. This time it is the variables addresses (references), which are copied to the stack, and the method *Swap()* knows that *a* and *b* must be interpreted as references to other variables. The result is, that it is the two variables in the *Main()* method which are changed.

Note that one can combine value parameters and reference parameters so that a method can have parameters for both reference and value transfer. Also note that a method can be overloaded for *ref* parameters, and the program *SwapValues* has two *Swap()* methods.

Methods have a type or they are *void*. The fact that a method is *void* means that it does not have a value, and it is therefore not required to have a *return* statement. As an example you have the method *Add()* in the class *Student*. A *void* method may well have an empty *return* statement, which then has the effect that the method is terminated. If a method has a type, it must have a *return* statement that returns a value of the same type as the method's type. It is important to note that a method can only return one value, but the type of this value in turn can be anything, including a class type or an array. As an example, the method *GetCourses()* in the class *Student* returns a *List<Course>*, and precise, it is a reference to such an object.

As can be seen from the above, a method's parameters are created when the method is called and initialized by the calling code. The parameters are removed again when the method terminates, and they live only while the method executes and they can only be used or referenced from the method itself. We say that the parameters scope are the method's statements.

The same applies to the variables as a method might create. They are called *local variables*. They are created when the method is called, and removed again when it terminates. Their scope is also limited to the method's statements. A local variable can be created anywhere in the method, but they are all created, however when the method is called, but if a variable is referenced by a statement before it is defined, the compiler fails. In conclusion, a method can refer to

1. instance variables in the methods class
2. parameters
3. local variables

where, the last two have their scope limited to the method itself. By contrast, the scope of an instance variable is limited to the object, so that the variable live as long as the object does.

9.5 OBJECTS

Seen from the programmer an application consist of a family of classes, but from the running program it consists of a family of objects that are created on basis of the program's classes. A class is a definition of an object in the form of instance variables that defines which data the object must consist of, as well as methods that defines what can be done with an object. If you have a class such as *Subject*, you can define a variable whose type is *Subject*:

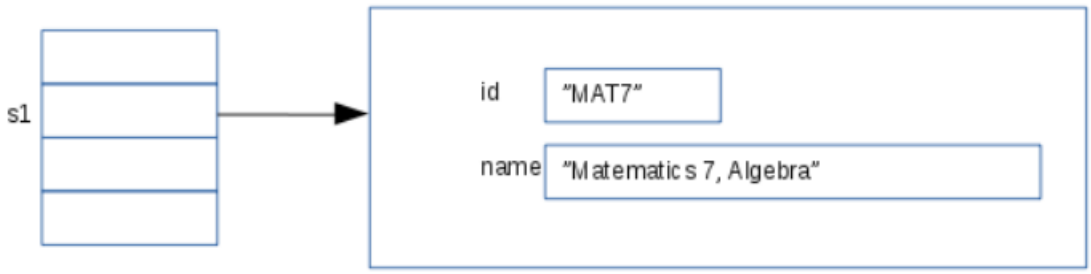
```
Subject s1;
```

s1 is a variable as all the other variables, but it has not (yet) a value. The variable can contain a reference that you can think of as a pointer that can point to (refer to) an object. If the variable does not refer to an object, its value is *null*, which merely indicates that it has no value. Objects are created with *new* as follows:

```
s1 = new Subject("MAT7", "Matematics 7, Algebra");
```

That is *new* followed by the class name and a parameter list that matches the parameters of a constructor. In this case, the class has a constructor that takes two *String* parameters, and an object can be created, as shown above. Sometimes we say that the statement *instantiates* an object. This means that when the variable *s1* is defined, a variable is created on the stack:

and after the object is created, the picture is the following:



When an object is created, the class's constructor is performed, and if there are several, it is the constructor whose parameters matches the parameters transferred with *new*. This means that the place allocated to the object's instance variables typical are initialized with values in the constructor. In fact, the above image is not quite correct, for a *String* is an object, and the two instance variables should therefore be shown as pointers to *String* objects. I have not done that partly because strings in an application in many ways are used as if they were primitive values, and partly because the drawing better matches the way you think of a *Subject* object.

The object is created on the heap, which is a memory area in which the run-time system can continuously create new objects. When creating an object, the heap manager allocates room for object's variables, then the constructor is executed. The object then lives as long there is a reference to it, but when it is no longer the case, either because the variable that references the object is removed, or manually is set to *null*, so the object is removed by the garbage collector, and the memory that the object applied, is released and can be used for new objects. The garbage collector is a program that runs in the background and at intervals remove the objects that no longer have references.

9.6 VISIBILITY

Visibility tells where a class or its members may be used. As for classes, it's simple, when a class is defined either *public* or else you specify no visibility. A *public* class can be used anywhere, and any other class can refer to a *public* class. However, if you do not specify any visibility, the class can only be used by classes in the same *namespace*.

Regarding the class's members, there are several possibilities as I will explain later, but for the moment is it enough to know *public* and *private*. A *private* member of a class can only be referenced inside the class, while a *public* member can be referenced from other objects.

9.7 STATIC MEMBERS

Both variables and methods can be *static*. A static variable is a variable that is shared between all objects of a class. When creating an object of a class on the heap, there is not allocated room for static variables, but they are created somewhere in memory where everyone has access to them, and if they have *public* visibility, it is not only objects of the class, which have access to them.

As an example of using a *static* variable, the class *Student* has an *id*, which is a unique number that identifies a student. When you create objects, it is necessary that these objects can be identified by a unique key, and to ensure uniqueness, this number is automatically incremented by 1 each time a student is created. It is possible because the class has a *static* variable that contains the number of the last student created. You must note that it is necessary that this variable is *static*, since it would otherwise be created each time, you creates a *Student* object. It is, in this way of identifying objects using an auto-generated number, a technique, which is sometimes used in databases. In this context, the solution is a little searched when the number is not saved anywhere and students will then possibly get new numbers the next time the program is executed, and the purpose is indeed only to show an example of a *static* variable.

There are many other examples of static variables, and as an example I can mention the class *Cube*, which had a random number generator, which was also defined as a *static* variable. There were the reason that all *Cube* objects should use the same random number generator when it is initialized by reading the hardware clock. If each *Cube* object had its own random number generator, these would possibly be initialized with the same value, thereby generating the same sequence of random numbers.

Classes can also have static methods, and in fact I have already used many examples. As an example the method *StudentOk()* in the class *Student*. When a class has a *static* method, it can be used without having an object as the method can be referred to with the name of the class:

```
if (Student.StudentOk(”poul.klausen@mail.dk”, ”Poul Klausen”)
{
}
```

In general is a static method written as other methods, and can have both parameters and have a return value, and the same rules apply regarding visibility, but a static method can not reference instance variables, it is not associated with a specific object. You must specifically note that within the class where the method is defined, it can be referenced in the same way as any other of the class's methods.

A class may also have a *static initialize block* or static constructor that can be used to initialize static variables, for example has the class *Student* the following block:

```
static Student()
{
  nummer = 1000;
}
```

because by one reason or another I wants that the first student must have the key 1001. In this case there is no justification for the block, then you as well could initialized the variable directly, but in other contexts it may be important to initialize static variables otherwise than by simple assignment, for example by reading the data from a file. You should also note that the syntax is simple.

If you considers a main class, Visual Studio creates the following class:

```
using System;

namespace Students2
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

which alone has a method with the name *Main()*. When the program is executed, the run-time system search for a method with this name and where appropriate executes the method. As the run-time system does not create an object of the class *Program*, the method must be *static*. If the *Main()* method wants to execute a method in the same class, it must generally be *static*, and the same applies if you in a static method in the main class refers to the variables in the class:

```
using System;

namespace Students2
{
  class Program
  {
    private static Random rand = new Random();

    static void Main(String[] args)
    {
      Test00();
    }

    private static void Test00()
    {
      Console.WriteLine(rand.Next());
    }
  }
}
```

That's why I until this place has always defined members in the main class as *static*. If you do not want that, and it is not necessary, you must write something like the following:

```
using System;

namespace Students2
{
 class Program
  {
    private Random rand = new Random();

    static void Main(String[] args)
    {
      Program obj = new Program();
      obj.Test00();
    }

    private void Test00()
    {
      Console.WriteLine(rand.Next());
    }
  }
}
```

This means that the *Main()* method creates an object of the class itself, and using this object it can then refer to non-static methods where you can use ordinary instance variables. There are rarely any good reason for this step, and typically the main-class will consists only of static variables and static methods.

10 FINAL EXAMPLE: LOTTO

At the end of the first book I will show a slightly larger example, where I as mentioned in the foreword mainly will focus on how the program should be written and less on the code. For the code I refer to the finished program, which incidentally is fully documented.

What to learn from this project

1. How to write a program that uses more classes
2. How to write a classic command

The task is to write a program that can print lottery tickets on the screen, but a program that also can be used to from the week's lotto numbers to determine the number of correct rows. The program should be used as a command from a command prompt where the user with options specify what the program should do. A typical use of the program is that the program is used to create the rows you want to play. When the week's winning numbers are drawn, you then use the program to determine the game's outcome. There are several (many) kinds of lottery, examples includes a lottery with 36 numbers and a lottery with 48 numbers that are examples of Danish lottery games. The program must be written so that it can be used for all lottery games which is of that nature.

A lottery game consists of a number of lottery rows, where a row consists of a number of different lottery numbers that are integers within a range, and in this program, a lottery game is characterized by the following parameters:

-a	the minimum allowable number	1
-b	the maximum allowable number	36
-r	number of lottery numbers at a row	7
-n	number of rows to play	

where the last column is a default value that corresponds to the ordinary Danish lottery. These values can be obtained as options on the command line. You can also use the following options:

-o	name of the output file for the result, where the default is the screen
-i	name of the input file when the program is used to check the week's lottery
-u	the week's lottery numbers separated by spaces

The last two options are used to check the lottery numbers of the week, while the other five options are used to create lottery rows for a new game. The program is called *Lotto*, and below are examples of legal commands:

```
Lotto -n 20
Lotto -a 1 -b 48 -r 6 -n 20 -o uge42
Lotto -i uge42 -u 2 13 18 22 34 35
Lotto -i uge42 -u 2 13 18 22 34 35 -o uge42-res
```

Generally, there are the following requirements for the arguments on the command line:

- All options which have a default value (*a*, *b*, *r*, *o*) may be omitted.
- Options may appear in any order.
- There must be no conflicting options (for example *-r* and *-i*) on the same command line.

The concept of a lottery ticket is not included in the solution as it in this program does not make sense, a lottery game is basically one large coupon that can have any number of rows.

In terms of the result of lottery there are the following requirements:

1. The numbers in each lotto rows should be sorted in ascending order.
2. The same row must not occur several times.
3. The rows should be sorted in ascending order (after the first number, after the next number, and so on).
4. The first line in the result defines the game and contains the smallest and largest legal lottery number, and the number of values in a row and the week number.

With regard to the results of the verification it is a file (the screen by default) that contains all rows, but where each row is added a number indicating the number of correct values. The result is sorted by this number in descending order.

If you enter an illegal command, for example because there is wrong, missing or conflicting options, illegal lottery numbers, illegal files, etc., the program must terminate with an error message and show the program syntax. If checking the week's lottery and there is an illegal row, the number after the row should be replaced with a short error message, and possibly illegal rows should be added last in the result.

Boundaries for the program's parameters are:

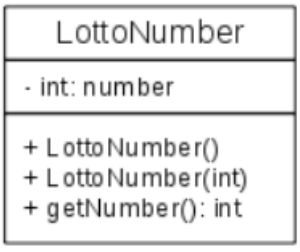
$$a \geq 1 \wedge a < 100$$
$$b \geq a + 5 \wedge b < 100$$
$$r > 2 \wedge r \leq 15$$
$$r < b - a$$

That is a lottery number maximum can have two digits, and a lottery row can have up to 15 numbers. In the same way there are some minimum requirements. There are really not many reasons for these requirements in addition to making it easier to formats the result and avoiding some special cases, such as the game can be empty.

10.1 DESIGN

The above can be seen as a specification of requirements for the program to be written. The program can be written in many ways, and from the one to use the program it plays no great role, as long as the program meets the requirements, but in practice, programs must be maintained, and here it is very important how the program is written. It is important that the program is written in such a way that the code is easy to read and understand, and it is important that the code is written so that a change in one place does not mean that large parts of the program must be rewritten. To meet these goals the program must be organized into good classes, and the question is what it is and how to find and write these classes. It is not simple, and there is not a unique solution, and this is largely what the following books is about, but below will explain a little about how this program is organized into classes, and thus the program's architecture.

If one considers the concept of a lottery row, then it is a sequence of different integers (lottery numbers) within a range. The lotto number is nothing more than an *int*, but I have nevertheless chosen to define a class to represent a lottery number. The reason is that the class must be able to validate a lottery number and check that it is legal in relation to the current lotto game, and the program should therefore have the following class:



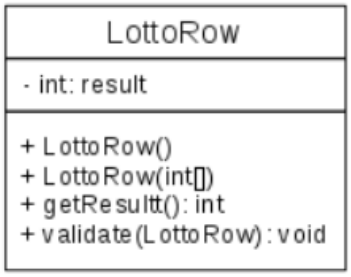
Often you will in system development illustrate the classes that way. It says that the program must have a class called *LottoNumber*, and this class must have a private *int* variable called *number*. In this case, this number represents the value of a lottery number. The figure also shows that the class should have two constructors, where the first is a default constructor that should create a *LottoNumber* object with a random legal value, while the other has the value as a parameter. It should be used when the program creates the week's lottery. In addition the figure shows that the class must have a method *getNumber()* to returns the lottery number's value.

The class *LottoNumber*, must as mentioned validate a lottery number and should therefore know what is legal values. In this case, it is solved by defining the following class

The class should have a method *init()* with three *int* parameters which indicates the minimum legal lottery number, the largest legal lottery number and the number of lottery numbers in a row. The class has methods that can return these values. In the figure, the methods are shown underlined, which means that they are *static* methods and can be used without having an object. The solution was chosen because the three values concerning a concrete lotto game must be available throughout the program. It should be emphasized that it is not the best solution, but it is the solution that is feasible with what until this place is said about C#.

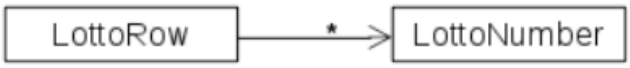
With this class available, it is clear that the class *LottoNumber* and its methods can validate whether a lottery number is correct.

The two classes *LottoNumber* and *LottoGame* is both simple, but the next class is more complex:



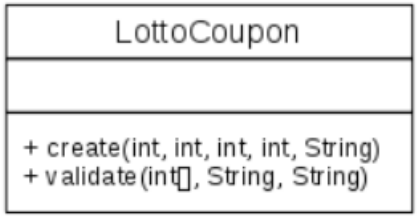
The class represents a lottery row, and thus a sequence of lottery numbers. The class has two constructors, where the first must create a lottery row with random lottery numbers, while the other creates a lottery row with specific numbers and should be used to represent the week's lottery. Both constructors must ensure that the lottery row created is legal in relation to the current lottery game, and therefore have to use the class *LottoGame*. In particular, is the first constructor complex since it must ensure the creation of a lottery row where all the numbers are different, and that the row's numbers are sorted in ascending order. The class should have a method *validate()* with a parameter that is a *LottoRow*. This parameter represents the week's lottery, and the method should be used to determine how many correct numbers there are on the current row. The result is that the method updates the *result* variable, and when it is private, there is also a method that can return the value.

The class *LottoRow* must consist of a number of *LottoNumber* objects, but instead of showing it as a variable in the class figure, we show it as a link between the two classes:



Here the * tells that a *LottoRow* object refers to several *LottoNumber* objects.

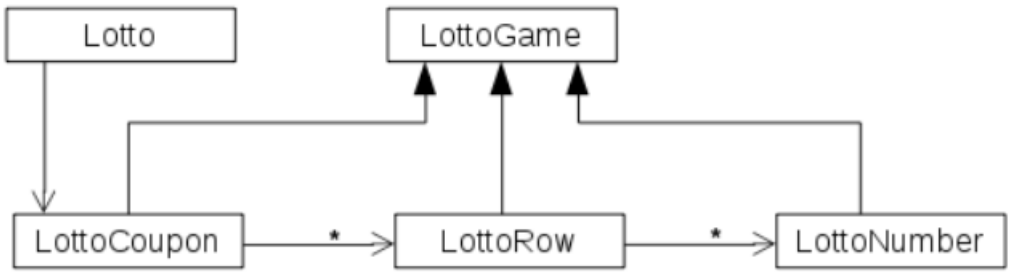
There is also a need for a class that may represent a number of lottery rows and thus a particular lottery game. Although I above has mentioned that the program not uses a concept similar to a lottery ticket, I have decided to call the next class for *LottoCoupon*, and one can think of the class as a concept similar to a lottery ticket of any size:



The class has only two methods, but they are, in turn, complex both. The first must create a new lottery game, and it has five parameters, where the first three are the game parameters (minimum lottery number, biggest lottery number and the number of lottery numbers in a row), next is the number of rows to be played, and finally the name of the file that will contain the result. With the class *LottoRow* available is in principle simple enough to create the wished number lotto rows, but the method must ensure not to create two identical rows, and the rows must be sorted before being written to the file. Finally, the method *create()* will write the rows to the file, and here it was decided that the first line should be the game's parameters. The method *validate()* is used to validate a lotto game against the weekly lottery. The method has three parameters, the first are the numbers for the week's lottery, while the next is the name of the file with the game to be validated, and the last is the name of the file to the result. The method then should both read data from a file and write data to a file. The most difficult is the of course validating of the individual rows. The class *LottoCoupon* is definitely the program's most complex class.

As the last is the class *Lotto*, which is the class with the *Main()* method. In principle, it is a simple method because it simply must create a *LottoCoupon* object and call one of the methods. Yet fills the code much, because the *main()* method must parse the command line, and this time there are many possibilities for the arguments. Finally, it is also the *Main()* method that has to print an error message in case of errors.

The program consists of a total of 5 classes, and the relationship can be illustrated by the following figure:



If you consider the above design you will notice that the design does not utilize that the language is C#. For example, method names are written with a lowercase letter as the first character and I have also not utilized that C# supports properties. In fact, at the time of design, it is intended to work independently of current programming languages.

10.2 PROGRAMMING AND TEST

I will not show the code for each classes here, but refers instead to the final solution. You are encouraged to study the code in detail and focus on how the methods are written and works, and there are many lines of code to address.

When a program is finished, it must be tested. I would in other books specifically look at the test, which is not all that easy again, but in this case you must test the software with different combinations of options and check whether you get the expected result, and it is important to test all combinations and also combinations that result in an error.

The program *Lotto* is as mentioned considerably larger than the examples which I otherwise dealt with in this book, but there is still talk about an example and of a program for its own fault, and it's not a program that has no practical justification. There is still a part of C# you have to know to be able to write programs with value in the real world, and especially lacks what it takes to write a program with a graphical user interface. It is the subject of the next book in this series.