C# 1

Introduction to programming and the C# language

Poul Klausen

bookboon.com
The eBook company

Poul Klausen

C# 1

Introduction to programming and the C# language

Download free eBooks at bookboon.com

C# 1 Introduction to programming and the C# language
© 2012 Poul Klausen & bookboon.com
ISBN 978-87-403-0250-9

Download free eBooks at bookboon.com

Contents

Deloitte.

Discover the truth at www.deloitte.ca/careers

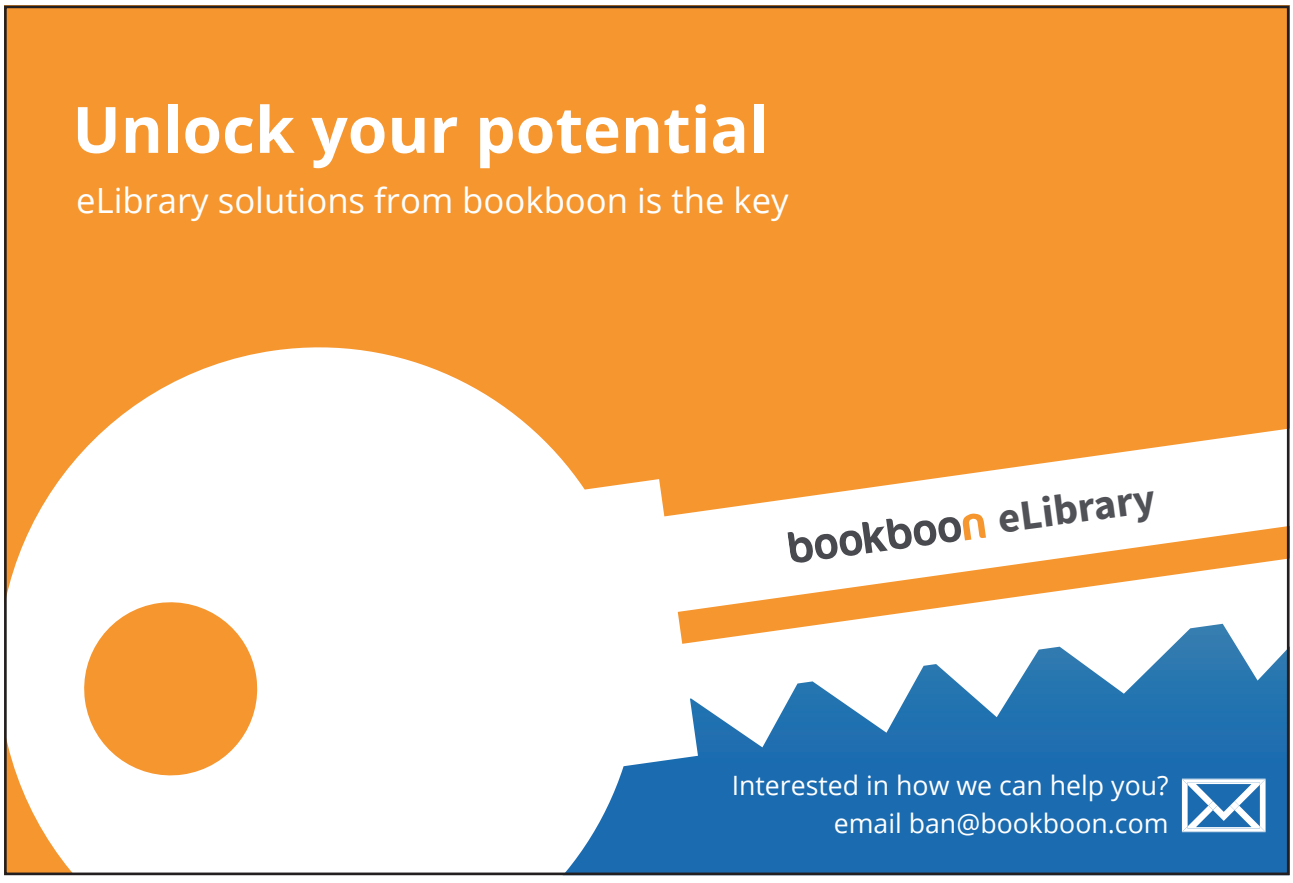
© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

Click on the ad to read more

Download free eBooks at bookboon.com

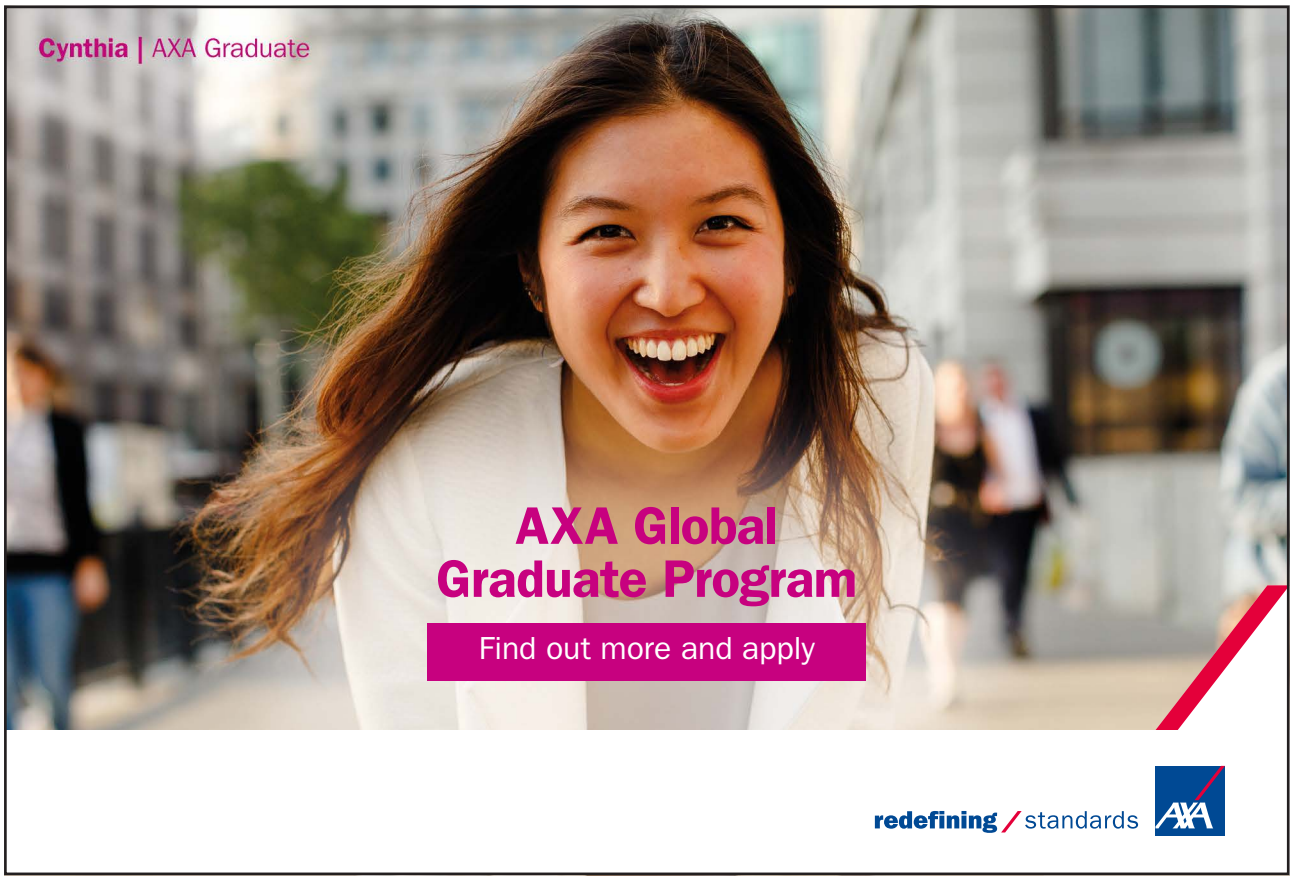


Click on the ad to read more

Download free eBooks at bookboon.com

Click on the ad to read more
Download free eBooks at bookboon.com

Click on the ad to read more
Download free eBooks at bookboon.com



Click on the ad to read more

Download free eBooks at bookboon.com

Click on the ad to read more
Download free eBooks at bookboon.com

Foreword

This book is the first in a series of books on software development for the .NET platform. The programming language is C#, and although the books thus focuses on the language C# and the selected platform, then programming in general play a large role, and the books has also focused on concepts such as algorithms, design and program quality. I have sought that each book must be read independent of each other, but the current book or similar substance may be regarded as a prerequisite for the subsequent, and that applies to some extent also C# 2.

The books is aimed at anyone who wants to deal with programming and the .NET platform, but because of selection of the examples the books are primarily intended as either teaching or supplemental materials in higher education. The books are not directed at any particular education, but it can be used in all courses which include courses in programming. Finally, the books could be used by professional software developers either as inspiration or as a reference regarding specific technologies.

The books have a practical purpose, so that the primary goal is to show how to do. Of course there are also more theoretical explanations, but I have tried to minimize the theoretical material in order to quickly reach what you need to write a program. Most of the material is presented through a large number of examples and the explanations, which are associated therewith, and the books are largely divided into corresponding to these examples. All examples have a bland name, that is name of the project for which they were created during the development tool, but in the book each example has a subtitle in the form of a word or phrase which briefly explains what the example shows, explains or deals with. There are references to these titles in the table of content.

All examples are developed using Visual Studio, but the tool is only considered to the extent necessary to be able to write programs using Visual Studio. Although the books may well be read with profit, if you use another development tool I would recommend that you have Visual Studio available. All the books examples are as complete Visual Studio projects for download from publisher's website. The projects related to a specific book are downloaded together as a zip file.

The current book is, as the title suggests, an introduction to programming and the language C#. The book requires no special assumptions of the reader and is aimed at anyone who wants or needs to learn about programming. It is thus not a prerequisite that the reader has knowledge of programming, but only that the reader is interested in programming and would have Visual Studio installed on his computer. All the book's examples are written using Visual Studio 11 beta.

The aim of the book is the language and programming in general and to consider the basics and what is necessary knowledge for being able to write programs. Therefore, all examples are simple console programs. When you have to give an introduction to programming, you can choose

Download free eBooks at bookboon.com

- only to look at console programs as a way to keep focus on the basics regarding substance of programming and the language
- quickly to introduce the necessary to be able to write programs with a graphical user interface and thus to arrive quickly to write more interesting and realistic programs

I chose the first way because the other has a tendency to obscure the basic and almost drown all the basic ingredients in the incredible number of concepts and details related to the development of a program with a graphical user interface. It is simply my experience that it is the right way, and what it takes to write Windows programs has got its own book. The price is that the examples in this book seems a little boring – depending on the eye of the beholder. You have to start somewhere, and I would recommend that you have the basics in place before tackling the more advanced topics. It should be added that you can easily work with the material in C# 3 after reading this book Parts 1 and 2.

The book is divided into 5 parts:

- The first part is a brief introduction to programming and C#. The goal is to introduce all the basic concepts without taking every detail. Stated slightly differently, the goal is that after part 1 you should be in a position to be able to write simple console programs.
- Part 2 deals with object oriented programming which masks the way to program today and the concepts associated with them. The substance of part 2 must be regarded as basic knowledge which should be in place, before you are able to develop complete applications in practice.
- Part 3 deals with collection classes that are part of every modern language. The book has only at a limited extent focus on the individual classes implementation, including the advantages and disadvantages, but focuses instead on how the classes are used. For a more detailed discussion of the classes characteristics, see C# 7.
- The book's fourth part deals with files. Files do not play the same role in practice programming as before, yet there are situations where it is necessary to work with files. The book focuses primarily on the treatment of text files and object serialization. If a program needs to deal with major external data volumes, it will in practice always be in the form of databases, and here refers to C# 4.
- Finally I am closing the book with part 5 as two slightly larger examples. Part 5 illustrates not new substances, and in order to continue reading the other books you can very well skip this part. The goal is to show the many concepts that are discussed in the book, in a slightly larger context, while also showing a little bit about how to work with application development in a larger perspective and in relation to issues that are more complex than it is in the book's other examples.

Poul Klausen

Download free eBooks at bookboon.com

Part 1 Introduction to C#

A computer program is a family of commands executed in a specific order that together solves a specific task. A program is written as a text document that contains all the necessary commands. This document is called the programs code or source code. The individual commands must be written in a very precise way, that the computer can understand them, and it is here a programming language comes into the picture. A programming language provides 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. Throughout this book the programming language C# is used, which is a widely used language in the Microsoft world.

As mentioned above, you write a program as a text document (in practice several or many) and the program is thus a simply document with commands. Commands are also called statements. Because these commands or statements are just text, the machine can not immediately execute the commands, but they must first be translated into an internal format that the computer understands. This process is called translation or compilation and executed by a program that can convert statements written in a particular programming language for the computer's internal commands. The program is usually called a compiler. During the translation the program is controlled for errors, and if there are errors, you get an error message and the error must be corrected before the program is translated anew. Not all errors are found during compilation, but only syntax errors that are errors where a statement is not written in accordance with the programming language rules. A compiled program can easily contain other errors, for example a wrong calculation.

To write a program, you naturally have to learn the programming language chosen, but you also must learn how to solve a task and formulate your solution using the language's statements. It is the latter that is the hardest, and there is rarely a unique solution. Solving a problem and formulating your solution by using a program language is also called writing an algorithm. Programming is therefore largely a matter of writing algorithms, something which I will return to repeatedly.

When you have to write a program, you need a tool that can be used for entering the program code, and in principle one could do that with Notepad and the compiler, but in practice you will always use a specific development tool, because it makes the job much easier. In the following I will use throughout Visual Studio, that is Microsoft's general development tool for a wide variety of tasks, including writing code in C#. It is an integrated package that contains all the tools necessary for the development of a number of different program types.

Download free eBooks at bookboon.com

1 Introduction

C# is an object-oriented programming language. The fundamental architectural element of a program is a class, and from a programmer's perspective is a C# program a family of classes, that collectively define all the application's properties and functionality. Writing a program is thus to define – design – and write the code for the program's classes. Nothing in C# exists outside a class. A program will also operate by many other classes that are not written by the programmer, but classes that are coming from the .NET framework, and thus is available to 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 must be written with a particular naming scheme, but it is almost the only formal requirements for the architecture of a C# program.

Exam01

Hello World

A good place to start with a new programming language is the classic Hello World program that just prints a text on the screen. This program has become a mandatory part of any exposition of a programming language. The program can be written as follows:

```
using System;
namespace Exam01
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Hello World");
      }
   }
}
```

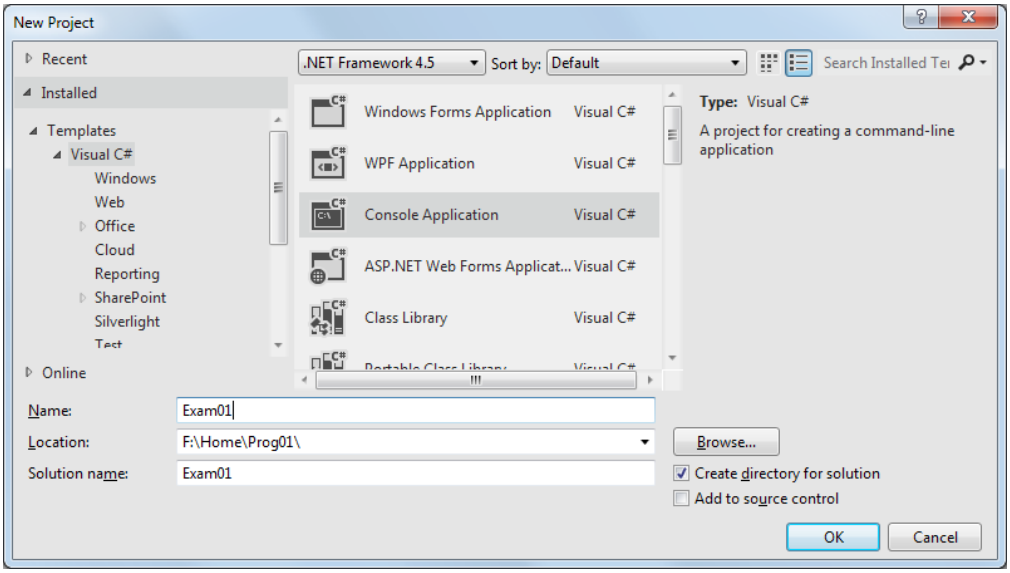
If you run the program the result is:

The program runs in a command window (prompt), where it prints the text *Hello World* on the screen. The program is not doing much, but it is a full-fledged program.

Download free eBooks at bookboon.com

How to

Open Visual Studio and choose *File* | *New* | *Project* from the menu:



Here you must be sure:

- that you have selected the language C#
- that you have selected the project type *Console Application*
- selecting the directory where to create the program files (here *F:\Home\Prog01*)
- that you have typed the program name (above *Exam01*)

Now when you click OK, Visual Studio will create a skeleton for an application:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Exam01
{
   class Program
   {
      static void Main(string[] args)
      {
      }
   }
}
```

Actually it is a full-fledged program that you can run on the machine – it made just nothing. You must write the program code, as shown in the introduction to this example. In this case, you only write a single line – a single statement:

Download free eBooks at bookboon.com

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

Thereafter, the program is finished and can be tested. From the menu you select

Debug | Start Without Debugging

Explanation

Note first 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 the class *Console*. When the program runs, there is nothing else than the *WriteLine* statement in *Main()* which print a text on the screen.

Click on the ad to read more

Download free eBooks at bookboon.com

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 automatically inserts 5 *using* statements in the code and thus 5 namespaces. In this case, only the *System* namespace is needed, and you are allowed to delete the 4 others (I have done that in the final version shown initially of the chapter). In principle there is no particular reason to delete unnecessary *using* statements as they do not have any bearing on the final program, so the only reason to delete them is to make the code more readable.

Comment

Visual Studio will automatically place the program in its own namespace, here called *Exam01*. 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 then forming 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 – in fact it is the only reasonable tool for developing .NET applications.

Download free eBooks at bookboon.com

2 Basic program architecture

The above example shows in principle the overall architecture of a C# program which is a class that has a *Main()* method as a starting point. The example was very simple, since the program consisted of only a single statement in *Main()*. In this section I will write a program where there are several statements, but also several methods. In this example, there is no special justification for splitting the code into methods – just to show how a method is called and written in C#. Methods are useful (necessary) for many reasons, but partly the methods can be used to subdivide the code into more manageable parts.

Exam02

Print a book

The goal is to write a program that on the screen can print information about a book

How to

Open Visual Studio and create in the same way as in *Exam01* a *Console Application* project. This time I have called the project *Exam02*, but otherwise all options are as above.

Visual Studio creates again a skeleton for a program, and the resulting code is shown below:

```
using System;

namespace Exam02
{
   class Program
   {
      static void Main(string[] args)
      {
         Title();
         More();
      }

      private static void Title()
      {
         Console.WriteLine("Vine fra Alsace");
         Console.WriteLine("Søren Frank");
         Console.WriteLine("ISBN: 87-7901-152-7");
         Console.WriteLine("Møntergården");
      }
```


Download free eBooks at bookboon.com

```
      private static void More()
      {
         Console.WriteLine("2. edition");
         Console.WriteLine("Published 2003");
         Console.WriteLine("179 pages");
      }
   }
}
```

The program can then be translated and run, and the result is a console window as shown above.

Explanation

In principle, it does the same as *Exam01*: It write text on the screen, just is the text in this example printed on several lines. In addition, the print statements are placed in methods that are called from *Main()*.

A method has – so far – the form:

```
private static void MethodName()
{
  // statemens
  }
```

Click on the ad to read more
Download free eBooks at bookboon.com

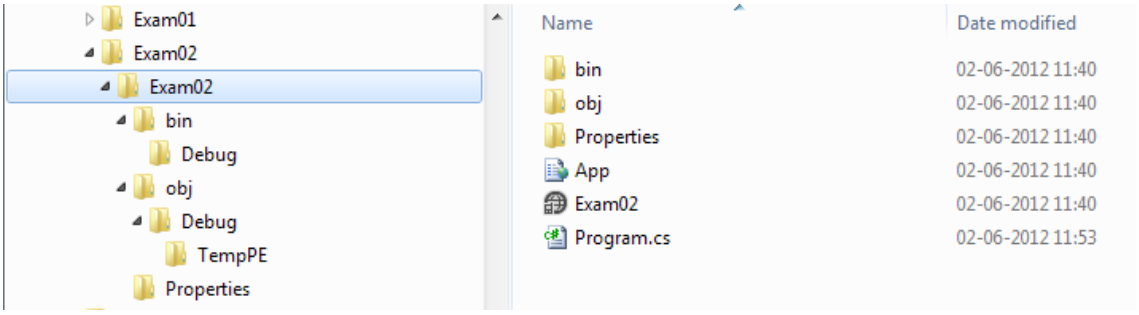
For example the method *Title()* consists of four statements that everyone writes a text. A method has a name, for example *Title()*, and it's the parentheses that tells, that it is a method. A method is called by typing its name. When the program starts, the two statements in *Main()* are executed, each of which calls a method.

Comment

It is obviously not a particularly interesting program because it every time print information for the same book. The program does not perform any data processing, but it comes in the next examples.

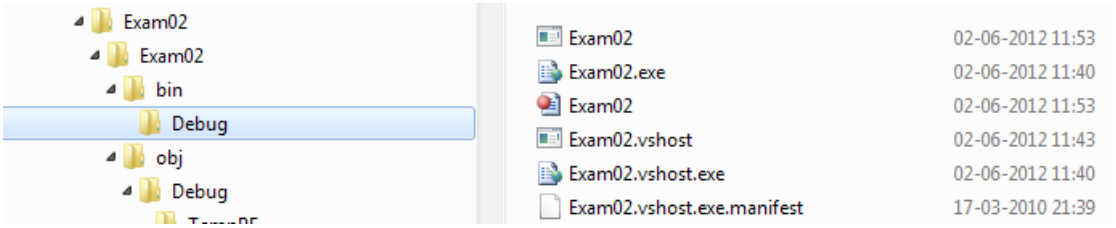
Comment

When you create a project in Visual Studio, it creates several directories and files:



The project create a directory, which is here called *Exam02*, including a subdirectory with the same name, and it is the directory that contains the project files. There are many files, but for a simple console application, there are only two that really is interesting for the programmer. The rest is for the sake of Visual Studio. The file *Program.cs* (see above) contains the program's source code, and hence the statements that the programmer has entered. Stated differently, it is the file you are working with in Visual Studio.

If you open the *bin* directory, and here the *Debug* directory you find the following files:



Here are the top the translated program called *Exam02.exe* (note that explorer by default does not display the extension *exe* and the second file is really called *Exam02.exe.config*). If you wish, you can take *Exam02.exe* file and copy it somewhere else (on the same machine or another machine), and the program can then be run by opening it in a prompt in the same manner as any other program.

Download free eBooks at bookboon.com

3 Variables

Applications must process data, and to do this they need a way to save or store the data. For this programs has variables, which may have or store a value. 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 regards to the naming of variables, but shall (should) be complied with:

- the name of a variable should always start with a small letter
- then there may follow any number of characters consisting of letters and digits
- a name must not contain spaces

If you follow these simple rules, you have never problems with names of variables, but some other characters are actually allowed.

Variables have a type that indicates which values can be stored in them, and how much a variable use of the machine's memory. The type also determines the operations that can be performed on a variable that is what can be done 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;

First you write the type, then the variable name, and finally assigned 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 otherwise you get an error in the translation.

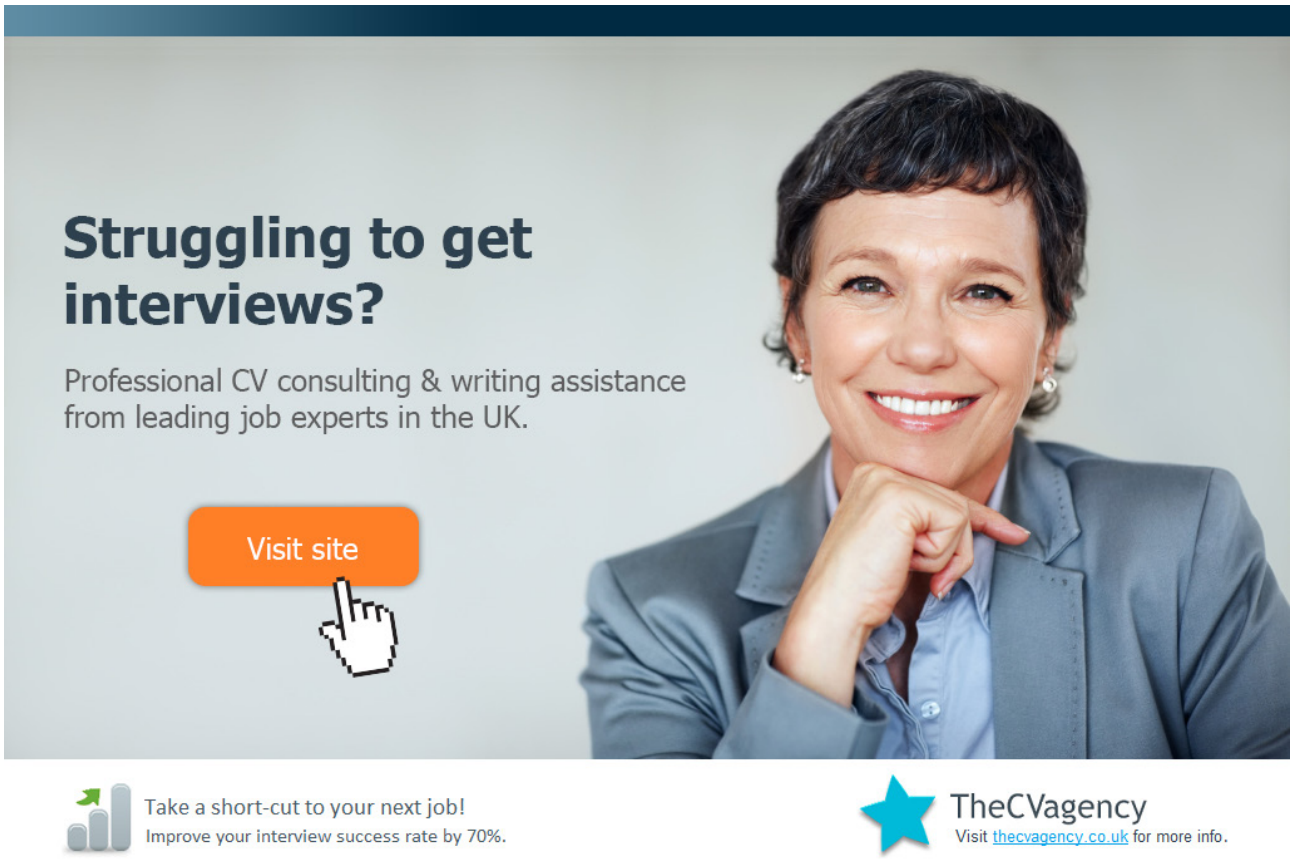
When the variables must be declared, it is because the compiler allocates space in the machine's memory, and that when the name appears somewhere in the code, the translator must know the name's meaning in order to check if the variable is used in a proper context. Is it not the case, the compiler give an error message. The program can only be tested when it is translated without error.

Download free eBooks at bookboon.com

C# has the following built-in 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.



Download free eBooks at bookboon.com

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.

Exam03

The sum of two numbers

Below is a program that calculates the sum of two numbers and prints the result:

```
using System;

namespace Exam03
{
   class Program
   {
      static void Main()
      {
         int num1 = 17;
         int num2 = 23;
         int sum = num1 + num2;
         Console.WriteLine("The sum of " + num1 + " and " + num2 + " is " + sum);
      }
   }
}
```

How to

In the same way as in the first two examples create a *Console Application* project in Visual Studio, and the code is entered as shown above. Then the program can be translated and tested:

This is a program that performs a data processing in the form of a calculation and is thus not simply a program that prints some text on the screen.

Explanation

First, the program declares two variables *num1* and *num2* that are initialized respectively with 17 and 23. The type is *int*, which means that the two variables may contain integer values. They are local variables, as they are created in the *Main()*, and they are only known in the *Main()* method. Then the sum of the two variables is stored in the variable *sum*. Note that the value stored in *sum*, is the result of an expression. *WriteLine()* writes the result. In this case, it builds a string from a number of parts or elements. Note that the individual elements are separated by + which here means string concatenating, and integer values automatically are converted to a string.

Download free eBooks at bookboon.com

Operators

C# has a number of operators, which acts on variables or values. The above program used the + operator. Note that the significance of the operator is dependent on the type of the variables or constants the operator acts on. In the first case where both operands are integers, the meaning is addition and in the second case (in the *WriteLine* statement), the meaning are string concatenating. Note that the above program also used the = operator, called the assignment operator and is used to assign a variable a value.

C# has the following operators in order of priority, and with decreasing priority downwards:

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

The individual operators are explained as they are used. The priority is of importance in expressions that involve multiple operators. The general rule is that you first evaluate the operators with the highest priority and in the case where there are several operators with the same priority they are evaluated from left. In for example the expression

a + b * c

b * c is calculated first, since * has higher precedence than +. If instead you writes

(a + b) * c

a + b are first calculated as parentheses have higher precedence than *. In most cases the use of operators are without much difficulties, but some operators requires a little explanation. Note particularly assignment operators, for example =+. For example means the following

```
int a = 11;
a += 2;
```


Download free eBooks at bookboon.com

that the variable *a* gets the value 13. Thus, it is just shorthand for the following:

```
int a = 11;
a = a + 2;
```

Another operator you should make special attention to is ++, which counts a variable up by 1. For example means

```
int n = 7;
++n;
```

that the variable *n* has the value 8. ++ may be written on both sides of the variable, and these can also be written as follows:

```
int n = 7;
n++;
```

The result in this case is the same. If the last statement, however, is included in an expression, it has significance on which side you write the operator. The rule is that if the operator is first (left), the variable incremented, after which the expression value is calculated, and is the operator after the variable the expression is calculated first, and then the variable is incremented. The result of the following statements

```
int n = 7;
int a = 0;
a = ++n;
```

Click on the ad to read more

Download free eBooks at bookboon.com

is such that *n* gets the value 8 and *a* gets the value of 8 (*n* is incremented by 1 and the result assigned to *a*), while the result of the following

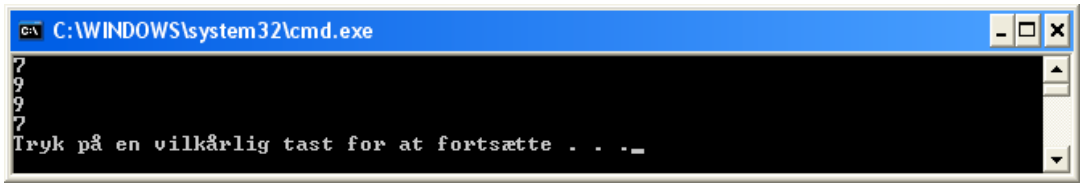
```
int n = 7;
int a = 0;
a = n++;
```

is that *n* gets the value 8 and *a* value of 7 (*a* is given value of *n*, and then *n* is counted up by 1).

Operator – operates in the same manner, but the value is decremented by 1. If you run the following example:

```
namespace Exam04
{
   class Program
   {
      static void Main(string[] args)
      {
         int n = 7;
         Console.WriteLine(n++);
         Console.WriteLine(++n);
         Console.WriteLine(n--);
         Console.WriteLine(--n);
      }
   }
}
```

you got the result:



Download free eBooks at bookboon.com

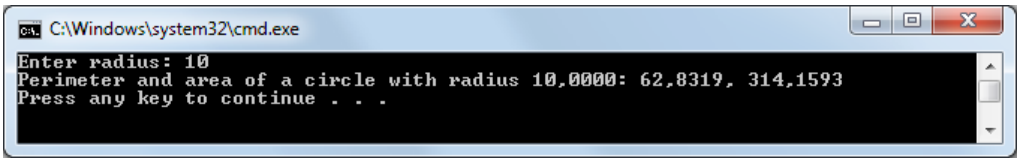
4 Console programs

As mentioned in the foreword this book treats only console applications. It is programs that are executed from a command line that writes the results to a text screen and the user can input data to a prompt. In this section I will look at how the user can enter data into the program and how to format the program's output.

Exam05

Perimeter and area of a circle

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:



How to

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);
}
```

Explanation

The program has in principle the same structure as the first example 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();
```

Note that the help text is written with the method *Write()*. The difference between this method and *WriteLine()* is that the *Write()* does not end with a newline. *ReadLine()* is a method in the class *Console* that collects user input until the user type *Enter*. Then the entries are returned as a string – a variable of type *string*. Note that *ReadLine()* always returns a *string*, and it is then the programs 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);
```


Download free eBooks at bookboon.com

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, there is an error handling. 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 is added, that is the value of the *r*. The next placeholder is called {1: F4}. It indicates that here the variable *p* is added – variable number 2 after the format string. F4 means that the value is added as a decimal number (F) with 4 places 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*.

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

Comment

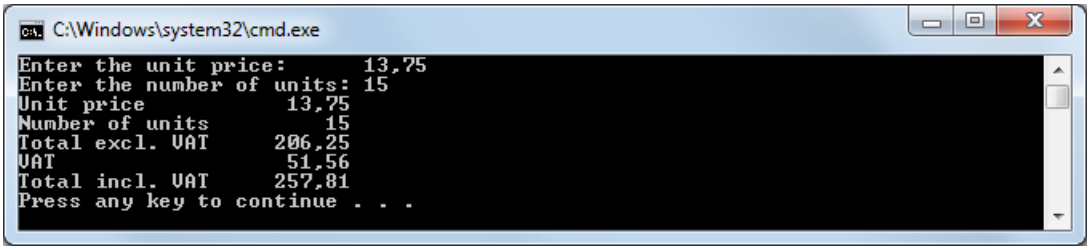
There are following options to format a placeholder:

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

Exam06

Product calculation

The next example performs a calculation. 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. If you run the program, the result could be the following:



How to

The starting point is again a Console Application project:

```
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);
}
```


Download free eBooks at bookboon.com

Explanation

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 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.

Exam07

Date and time

This program will print how much the time is:

The program will primarily show the formatting of the result, but also the use of type *DateTime*.

How to

The code can be written as shown below. Note that this time there are two methods that are called from *Main()*:

```
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());
}
```

Note that I did not show the whole code, but only the methods.

Download free eBooks at bookboon.com

Explanation

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* and which type is *DateTime*. Next, a method *Time1()* is called, where the variable *dt* is sent as a parameter. This means that it 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. Please 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.


Click on the ad to read more

Download free eBooks at bookboon.com

Comment

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 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.

Exam08

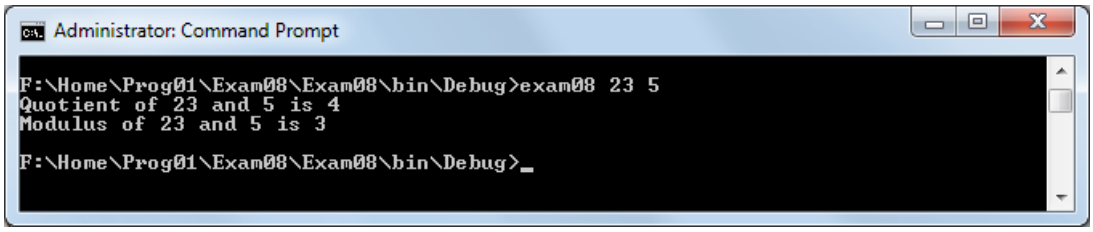
Arguments on the command line

When you create an application using Visual Studio, the *Main()* method has a parameter, that I have not used so far. It is used on the command line to transfer the arguments to a program. That is if you have a program called *Exam08.exe*, you can execute the program from the command line by typing

```
Exam08 a b c
```

where a, b and c are arguments to the program and are arbitrary strings separated by at least one space. There are no restrictions on the number of arguments and an argument need not be a single character as above, but may be any string. If the program does not do anything by the arguments they are ignored, and it is up to the program to address these arguments.

The example shows an application where the user must transfer two integers as arguments on the command line, and the program will then print out the quotient and modulus. The result could, for example be the following:



How to

The program is written in the same way as the other examples, and code for the *Main()* method is as follows:

```
static void Main(string[] args)
{
   long t1 = Convert.ToInt64(args[0]);
   long t2 = Convert.ToInt64(args[1]);
```


Download free eBooks at bookboon.com

```
    Console.WriteLine("Quotient of {0} and {1} is {2}", t1, t2, t1 / t2);
    Console.WriteLine("Modulus of {0} and {1} is {2}", t1, t2, t1 % t2);
}
```

Explanation

The *Main()* method has a parameter which is an array (arrays are discussed below), and which may represent arguments on the command line. Since the arguments from the command line are always strings, they are converted to integers (in this case of the type *long*) before the result can be calculated, which is the quotient and modulus of the two numbers. Note specially the operator % which is the modulus operator (remainder of division).

Note that if you run the program and do not specify two arguments, or if there is one of the arguments that is not an integer the program will crash with an error message. But if you specify more than two arguments, the last are just ignored.

This e-book
is made with
SetaPDF
SETASIGN
PDF components for PHP developers
www.setasign.com
Click on the ad to read more

Download free eBooks at bookboon.com

5 Program control

The above examples are all sequential so that the statements may be carried out in the order they are written. In practice, all programs need to make the execution of statements depending on a condition that occurs when using control statements. C# has the following fundamental statements for program control

- *if*
- *while*
- *do*
- *for*
- *switch*

Control statements are best illustrated through examples in the form of simple methods, which are the subject of this section.

Control statements are used in methods in the same manner as the other statements, for example *WriteLine*, but a control statement makes the execution of one or more other statements depending on a condition. Thus, it needs to refer to multiple statements as a whole, which is done by means of a block, which is just a number of statements in brackets:

```
{
    statement1;
    statement2;
    ....
}
```

For example are the statements in a method a block, but it will be apparent hereinafter, that one can have blocks within a block. You've actually already seen examples, where a *class* is also a kind of block which instead of statements contains methods.

Control statements are needed to be able to write programs that do something interesting. First with control statements available, you can begin to work on algorithms and hence write programs that solve a specific problem. The following will therefore also to some extent focus on algorithms.

if

An *if* statement has the form:

if (condition) block

Download free eBooks at bookboon.com

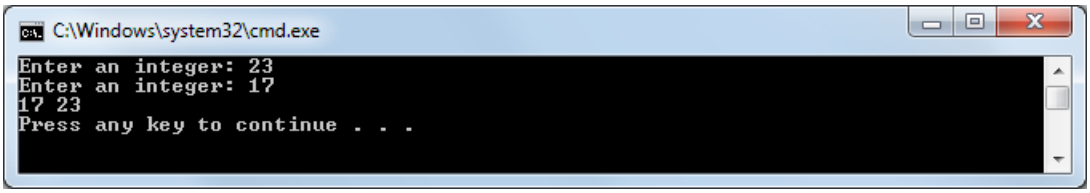
and the meaning is that if the condition is true, then perform the block (the block's statements). Otherwise, nothing happens. A condition is an expression whose value is a *bool* and hence an expression that is *true* or *false*. If only a single statement has to be controlled by a condition, you must omit the parentheses and simply write:

if (condition) statement;

Exam09

Sort two numbers

The task is to write a program where the user can enter two integers. The program will then print the two numbers in ascending order. An example of an operation of the program might be:



How to

The task can be decomposed into two sub problems:

- Enter two integers
- Print the numbers in ascending order

To solve the first problem one has to perform the same operation (enter an integer) twice, and it is therefore worthwhile to write this operation as a method:

```
static int Enter()
{
   Console.Write("Enter an integer: ");
   string text = Console.ReadLine();
   return Convert.ToInt32(text);
}
```

Note that the method has a type, but it is explained below.

To print the results you have to find the smallest number, and it is here the *if* statement comes at the track. I will use the approach that if the first number is greater than the last, I will swap the two numbers, and in one way or another I have in C# to determine (test) if the first number is greater than the other. That's exactly what an if statement is used to.

Download free eBooks at bookboon.com

One can define a method for the printing of the results follows:

```
static void Sort1(int a, int b)
{
   if a > b then
   {
      save a in a help variable t
      a = b
      b = t
   }
   Console.WriteLine("{0} {1}", a, b);
}
```

Strictly speaking it is not a method, but it is a solution formulated by using an informal language – it is an algorithm. The task is therefore to write this algorithm in C#:

```
static void Sort1(int a, int b)
{
   if (a > b)
   {
      int t = a;
      a = b;
      b = t;
   }
   Console.WriteLine("{0} {1}", a, b);
}
```

Click on the ad to read more

Download free eBooks at bookboon.com

Then the program itself can be written as follows:

```
static void Main(string[] args)
{
   int a = Enter();
   int b = Enter();
   Sort1(a, b);
}
```

Explanation

The method *Sort1()* has a condition in the form of an *if* statement that tests whether the value of *a* is greater than the value of *b*, and when it is true, the next block, which reverses the two numbers, is executed. Since the block is only executed if the value of *a* is greater than the value of *b*, the value of *a* will always be less than the value of *b* after the block is executed. The result is that the two numbers are printed in ascending order.

You should note how to write a condition in an *if* statement, and that it is an expression whose value is either *true* or *false*. It is an expression whose type is *bool*. Also note that the expression should be in parentheses.

You should also note the method *Enter()*, which is a method with a return value. Return values are addressed later in the section on methods, but until then you can think of a return value as a value attached to the name of the method with a return statement. That way you can get the value transferred to the place where the method is called such by writing:

```
int a = Enter();
```

This means that the value attached to the name of the method by return is stored in variable *a*.

Comment

A method's statements are an algorithm or a solution. The biggest challenge in writing programs is to learn to write algorithms, because a given problem can often be solved in several ways. Above I have shown an algorithm to swap two numbers and it can often be a good idea to start writing an algorithm in an informal language, because in this way frees the solution from the many details of the programming language and thus can focus on the problem itself and how it is solved. Once you have formulated the algorithm in an informal language, it is typically an easy task to translate the algorithm to the specific programming language, which here is C#.

Note especially the algorithm to swap the two numbers. It is a simple algorithm, but it's really important and it is an algorithm that you will meet many times.

Download free eBooks at bookboon.com

if-else

An *if* statement can be combined with an else part:

if (condition)
    block_1
else
    block_2

where both *block_1* and *block_2* can be simple statements. The significance is that *block_1* is executed if *condition* is true, and if not then *block_2* is executed. As an example, the preceding method that prints two numbers in ascending order can be written in the following way:

```
static void Sort2(int a, int b)
{
   if (a < b)
      Console.WriteLine("{0} {1}", a, b);
   else
      Console.WriteLine("{0} {1}", b, a);
}
```

Note also that the method can be written as follows:

```
static void Sort2(int a, int b)
{
   if (a < b)
      Console.WriteLine("{0} {1}", a, b); else Console.WriteLine("{0} {1}", b, a);
}
```

and as another example could be written as:

```
static void Sort2(int a, int b)
{
   if (a < b)
   {
      Console.WriteLine("{0} {1}", a, b);
   }
   else
   {
      Console.WriteLine("{0} {1}", b, a);
   }
}
```

Seen from the machine, the three versions are equally good, and the choice is solely a matter of what you think is most readable.

Download free eBooks at bookboon.com

Exam10

A quadratic equation

I'll show a program that can solve a quadratic equation, and therefore a program that solves a classical task from school and mathematics teaching. A quadratic equation is expressed as:

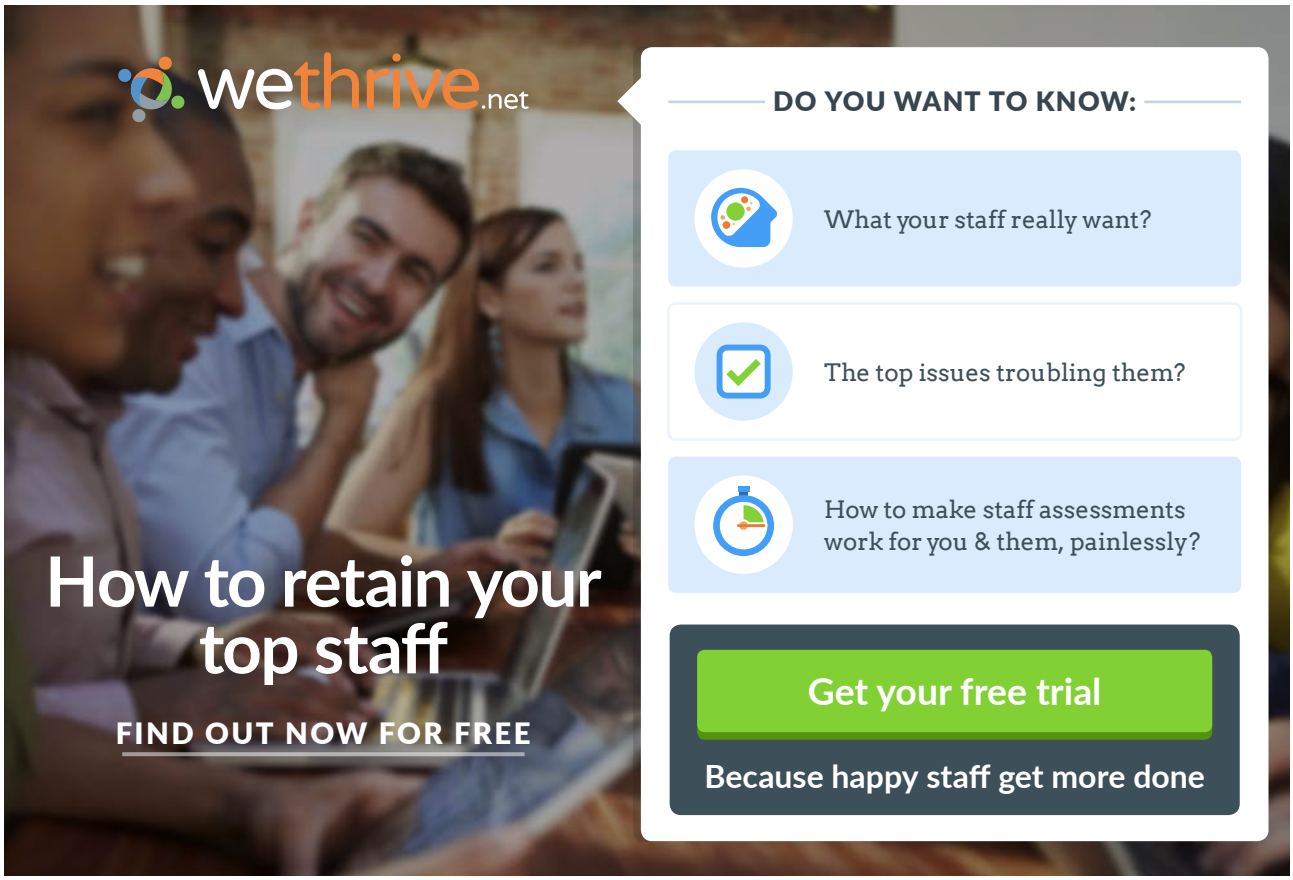
$$ax^2 + bx + c = 0$$

The solution formula is:

Given the discriminant : $d = b^2 - 4ac$:

$$\text{Solution: } \begin{cases} no\ solutions & if\ \ d < 0 \\ \dfrac{-b}{2a} & if\ \ d = 0 \\ \dfrac{-b \pm \sqrt{d}}{2a} & if\ \ d > 0 \end{cases}$$

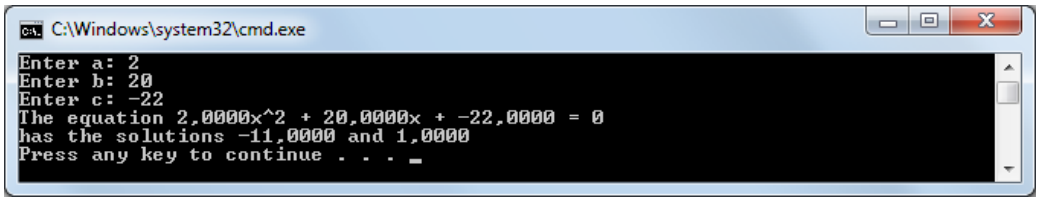
The task is to write a program where the user must enter the equation's coefficients (that is *a*, *b* and *c*). The program will then determine the equation's solutions using the above formula and print the result on the screen.



Click on the ad to read more

Download free eBooks at bookboon.com

An example of an operation of the program might be:



How to

The solution of the equation can be informally described as follows:

```
calculate the discriminant d
if d < 0 then no solutions
else if d > 0 then calculate two solutions
else calculate one solution
```

Immediately there is not much solution in that, but it breaks down the task into three sub problems, each of which is simpler than the original problem. This kind of problem decomposition is a principle that recurs in many tasks and is an important step towards a complete solution. Each of the three sub-problems is relatively simple, and the program can be written as follows:

```
class Program
{
   static void Main(string[] args)
   {
      double a = Enter("Enter a");
      double b = Enter("Enter b");
      double c = Enter("Enter c");
      Solve(a, b, c);
   }

   static double Enter(string text)
   {
      Console.Write(text + ": ");
      string line = Console.ReadLine();
      return Convert.ToDouble(line);
   }

   static void Solve(double a, double b, double c)
   {
      double d = b * b - 4 * a * c;
      Console.WriteLine("The equation {0:F4}x^2 + {1:F4}x + {2:F4} = 0", a, b, c);
      if (d < 0) Result();
      else if (d > 0) Result(a, b, d);
      else Result(a, b);
   }

   static void Result()
   {
      Console.WriteLine("has no solution");
   }

   static void Result(double a, double b)
   {
      Console.WriteLine("has the solution {0:F4}", -b / (2 * a));
   }
```


Download free eBooks at bookboon.com

```
   static void Result(double a, double b, double d)
   {
      double y = Math.Sqrt(d);
      Console.WriteLine("has the solutions {0:F4} and {1:F4}",
         (-b - y) / (2 * a), (-b + y) / (2 * a));
   }
}
```

Explanation

Note first the method *Enter()* which is substantially the same as in the previous example. There are two differences. The guiding text that tells the user what needs to be transmitted, are this time sent as a parameter. The second difference is that the method this time converts the input to a *double*, and the method must also return a *double* – the method's type is *double*.

The method *Solve()* solves the equation, and it performs three things:

- calculates the discriminant
- print the equation
- implement the above algorithm, which divides up into three sub problems, and here you mainly observe how the *if-else* statements is used

The three methods to print the result does not require much explanation, but note the last, and how to determine the square root of a number. This is done by the method *Sqrt()*, which is a method in the class *Math*.

Comment

The method *Solve()* includes *if-else* statements:

```
if (d < 0) Result();
else if (d > 0) Result(a, b, d);
else Result(a, b);
```

and when each condition control only a single statement, I have used, that it is not necessary to place the statement in a block. It's something you can discuss, and many will prefer to write the code in the following way, as they think it gives a more readable code:

```
if (d < 0)
{
   Result();
}
else if (d > 0)
{
   Result(a, b, d);
}
else
{
   Result(a, b);
}
```


Download free eBooks at bookboon.com

Seen from the finished program and the machine it is irrelevant, and the two codes are translated into the same and the one is neither more nor less effective than the other. The choice is the programmer's and is only a question of readability, and you should simply choose the version that you think is most readable. I think the first version is the most readable, but it's far from all who agree in that.

That a program is readable is actually more important than that, and here one must bear in mind that programs often require maintenance by anyone other than the one who originally wrote the program, and for it to be possible, it should be easy both to read and understand the program.

while

It is often needed to carry out a statement or a block several times until a condition occurs. Here you can use a *while* statement, which has the following form:

while (condition)
block

The significance is that the block and its statements are performed as long as the condition after *while* is true. Then the program continues with the next statement after the *while* construct.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

Exam11

The sum of the positive number less than 100

As an example is shown a program which determines the sum of all positive integers less than or equal to 100, that is the sum $1+2+3+...+100$.

How to

The program can be written as follows:

```
class Program
{
   const int N = 100;

   static void Main(string[] args)
   {
      long s = 0;
      int n = 1;
      while (n <= N)
      {
         s += n;
         ++n;
      }
      Console.WriteLine(s);
   }
}
```

Explanation

Note first that in the beginning of the program there is defined a constant, which is the largest number to be included in the sum:

```
const int N = 100;
```

It is not necessary, but it makes it easy to modify the program if, for example instead it must determine the sum of the numbers from 1 to 1000.

Note the algorithm. First define a variable *s* to the result. Next, define a variable *n* to the number to be added to the sum and initialize it to 1, which is the first number. Then repeat the subsequent block as long as *n* is less than or equal to 100 (the constant *N*). In the block two things happens: The value of the variable *n* is added to *s* and *n* is incremented by 1. Note that the last statement means that *n* is 1 more for each repetition, and thus the condition for the *while*, sooner or later becomes false.

Download free eBooks at bookboon.com

Comment

Initially it may be difficult to see through an algorithm as above and see that it does the right thing and here it may be a good idea to implement a desktop test where you try to manually keep track of what happens to the variables. For example, suppose that *N = 10*. The program must determine the sum $1+2+3+...+10=55$.. When the algorithm is performed the following happens, where the first column shows what happens to the variable *s*, and the second column shows what happens variable *n*:

s	n	
0	1	The two variables after they are initialized (created)
1	2	Variables after the 1. *while* statement is executed
3	3	Variables after the 2. *while* statement is executed
6	4	Variables after the 3. *while* statement is executed
10	5	Variables after the 4. *while* statement is executed
15	6	Variables after the 5. *while* statement is executed
21	7	Variables after the 6. *while* statement is executed
28	8	Variables after the 7. *while* statement is executed
36	9	Variables after the 8. *while* statement is executed
45	10	Variables after the 9. *while* statement is executed
55	11	Variables after the 10. *while* statement is executed

This means that the loop stops when *n* is 11, and at that moment, the variable *s* has the value 55.

Is it a large program that has many variables and iterations with many repetitions, the above is obviously not a viable option, but in simpler cases, it may be a good way to convince itself that an algorithm works – and to find a fault.

Visual Studio provides a better opportunity of using the Debugger, but I show it later.

Comment

The *while* statement is also called a loop, an iteration or a repetition. If the block contains only a single statement, you can in the same manner as for *if* omit the parentheses. For example the above program can be written as:

```
static void Main(string[] args)
{
   long s = 0;
   int n = 1;
   while (n <= N) s += n++;
   Console.WriteLine(s);
}
```


Download free eBooks at bookboon.com

Note it here is important that ++ stands after *n*, because *n* must be incremented by 1 after the calculation is done.

Again it is up to you and readability, if you will omit the parentheses or not.

Comment

The task is to determine the sum of a row of integers $1+2+3+...+N$, and above, it is solved by means of a loop. Seen in relation to programming it is perhaps the most obvious solution, but there are other options, since the sum is an example of a difference serie, and mathematics has a formula to calculate this sum. I mention this only to point out that a giving problem often can be solved in several ways, and it is not always the most obvious solution that is the best.

for

The *for* statement has three parts separated by semicolons: initialization, condition and expression:

for (initialization*; condition; expression*)
block;

Each of the three parts of the *for* statement can actually be empty. When the statement is carried out, the following occurs:

Click on the ad to read more

Download free eBooks at bookboon.com

1. Perform the initialization part (if there is one)
2. Test whether the condition is true (if the condition is empty it is true)
3. If the condition is true executes the following block
4. Otherwise the *for* statement is interrupted
5. Execute the expression part
6. Continue with step 2

Note that the initialization part is done only once. Note also that the block can be a single statement.

Exam12

Sum of positive integers

This example has the same purpose as above, but this time I will solve the task using a *for* statement.

How to

With use of the *for* statement, the program can be written as follows, where the constant *N* is as above:

```
static void Main(string[] args)
{
   long s = 0;
   for (int i = 1; i <= N; ++i) s += i;
   Console.WriteLine(s);
}
```

Note that the code resembles the previous example.

Explanation

The *for* statement have a counter – here the variable *i* – which counts the number of iterations. For each repetition the following statement is executed

```
s += i;
```

Note that the counter *i* is incremented by 1 after each pass, and that repetition continues until *i* reaches an upper limit.

Comment

The *for* statement is very flexible. The initialization part is usually used to initialize variables, and it is permissible to have multiple initializations when separated by commas:

```
for (a = 2, b= 3, c = 5; ...
```


Download free eBooks at bookboon.com

When, as above a variable is declared in the initialization part, the variable is not known outside the *for* statement and the corresponding block. You can declare multiple variables in the initialization part:

```
for (int i = 0, j = 1; ...
```

but they must have the same type. As for the condition there is no other requirement than it should be a legal boolean expression. That can include boolean operators, function calls, etc. The expressions section is typically used to control a controlling variable in the condition, but generally it can be anything that is an expression and also a function call. The expression section can have multiple expressions if they are separated by a comma, for example:

```
for (...; ...; ++i, j += 7)
```

In the above cases there is no advantage in applying *for* instead of *while*, and seen from the program, it has seldom the great importance of which loop structure is used. The choice is usually determined by the nature of the task, where one of the structures may be more appropriate to the other. However, *for* is the most flexible and the most frequently used.

do

There is another variant of the *while* statement, called a *do* loop that has the form:

do
{
statement;
...
}
while (condition);

The significance is that the statements in the block are performed and then the condition is tested. Is it true, repeat the block and it continues until the condition becomes false. Compared with the *while* statement is the difference that *do* always will perform the block at least once, since the condition is first tested after the block is executed. The *do* loop is not used as often as the other loops.

switch

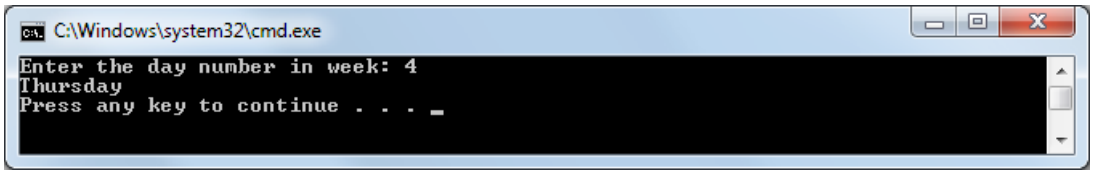
while, *for* and *do* are control structures for loops or iterations, while *if* is a control structure for branching or selection. In most cases, *if* is used for selections, but there is an alternative, called *switch*. It is best explained by an example.

Download free eBooks at bookboon.com

Exam13

Weekday

The task is to write a program where the user must enter an integer. If the number is 1, 2, 3, 4, or 5, the program will print the name of the corresponding day of the week, if the number is 6 or 7, the program will print the text *Weekend*, and otherwise the program will print an error message. Below is an example of a running program:



How to

The program can be written as follows, where in *Man()* the user must enter the number for the day of the week:

```
static void Main(string[] args)
{
   Console.Write("Enter the day number in week: ");
   string text = Console.ReadLine();
   switch (Convert.ToInt32(text))
   {
```

Unlock your potential
eLibrary solutions from bookboon is the key
bookboon eLibrary
Interested in how we can help you?
email ban@bookboon.com
Click on the ad to read more

Download free eBooks at bookboon.com

```
      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;
   }
}
```

Explanation

After the word *switch* is an expression:

```
switch(Convert.ToInt32(text))
```

The construction also has a number of *case* entries where the case is a constant followed by a colon. The constant must have the same type as the expression:

```
case 1:
```

When the statement is executed, the expression is evaluated, and the control is transferred to the case entry that is identical to the value of the expression. The statements after the case entry are then executed, until you meet a *break*-statement, a *return* statement or the *switch* statement ends. If there is no case entry, which corresponds to the value of the expression after *switch*, there are two possibilities. Has the *switch* statement a *default* entry, the control is transferred to that entry. Otherwise the entire *switch* statement is skipped. Note that a *switch* does not need to have a *default* entry.

In the above if the input is 6 or 7, the program in both cases the print

```
Weekend
```

If, for example the user enters 6, the program will continue with the statement after the case 7, as there is no break after the case entry for 6.

A *switch* statement is kind of multi-branching, and it can sometimes be a reasonable solution, but is not as often used as the other control statements.

Download free eBooks at bookboon.com

Exam14

The cross-sum

To conclude the use of control statements from this section, I will show two additional programs.

The first program calculates the cross-sum of a number, the user enters. If the user enters 123456 the cross-sum is 3:

$$1 + 2 + 3 + 4 + 5 + 6 = 21$$
$$2 + 1 = 3$$

The program will show how one can have two *while*-statements inside each other, but is also an example of an algorithm that are bit harder to solve than other algorithms shown so far.

How to

Solving the task can be decomposed into three sub problems:

- Enter the number
- Determine the cross-sum
- Print the result

Here the solution of the first and last problem are what I have done previously, and are simple problems, and the task is reduced to determine the cross-sum of a number. This task can be described as follows:

- determine the sum of the number's digits
- repeat until the sum is less than 10

A little more formally the task can be formulated as an algorithm in the following manner:

```
while number > 9 do
{
        let sum = sum of the numbers digits
        number = sum
}

number is now the cross-sum
```

The problem can be solved with a loop, but yet it is not entirely clear how to determine the sum of the digits of a number. This can be done with a second loop:

```
while number > 0 do
{
     add number % 10 to sum
     divide number by 10
}
```

where the first operation in the loop adds the last digit of the number to the sum, while the second operation throws the last digit off.

Download free eBooks at bookboon.com

With these considerations in place, you can write a method that determines and returns the cross-sum of a number:

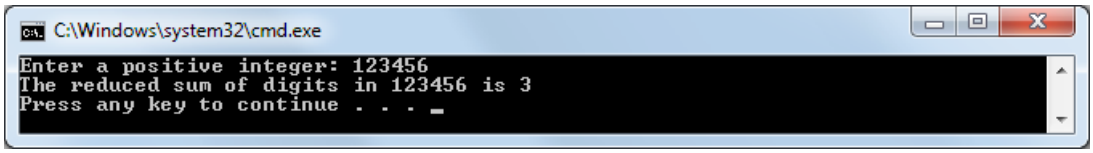
```
static uint DigitSum(ulong n)
{
   while (n > 9)
   {
      ulong s = 0;
      while (n > 0)
      {
         s += n % 10;
         n /= 10;
      }
      n = s;
   }
   return (uint)n;
}
```

The program itself can now be written as, where the method *Enter()* is as before:

```
static void Main(string[] args)
{
   ulong number = Enter();
   Console.WriteLine("The reduced sum of digits in {0} is {1}",
      number, DigitSum(number));
}
```

Click on the ad to read more
Download free eBooks at bookboon.com

If you run the program the result could be:



Explanation

The inner loop determines the sum of the numbers digits. Note the operator % which is the modulus operator. For example means

```
number % 10
```

residue by dividing by 10, which is the rear digit of the number. Note also the result of

```
number /= 10;
```

where the division by 10 (integer division) "throw" the last digit off. The outer loop is repeated as long as the sum of digits is greater than 9 – it is more than one digit.

Note also the data types. The parameter to the method *DigitSum()* has the type *ulong*. I chose a long to work with large numbers, and here again an ulong, because the number should not be negative. Similarly, the method returns an *uint*, which is a non-negative integer. Unsigneds are used less often, and I've mostly used them here to show how they are used. *n* is an *ulong*, and the method returns an *uint*. In the *return* statement is thus a need to put a large number (an *ulong*) into a smaller (an *uint*). You can't just do that without telling the compiler that you mean the business. You do that with a type cast:

```
return (uint)n;
```

which tells the compiler that it should convert *n* to an *uint*.

Exam15

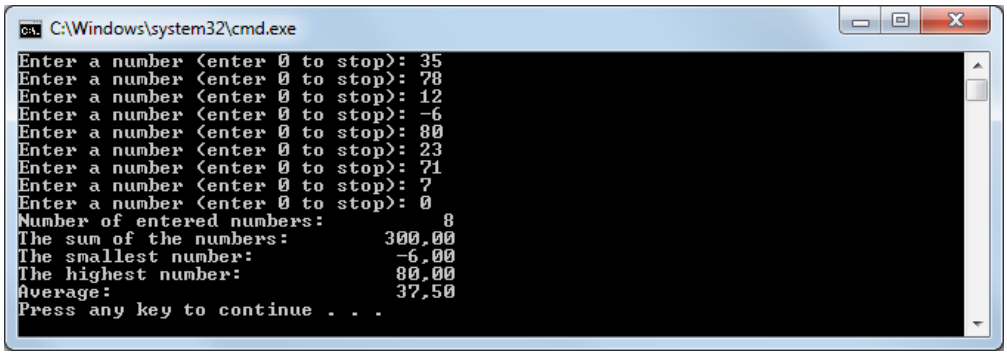
The biggest and the smallest number

This example is a program that uses the control structures *while* and *if*. *if* is the most frequently used control statement at general, and in practice, one can't imagine a program that does not apply *if*. In the program you must enter an arbitrary number of numbers and the method then prints

- number of entered numbers
- the sum of numbers entered
- the smallest of the numbers entered
- the largest of the numbers entered
- the average of the numbers entered

Download free eBooks at bookboon.com

An example of an execution of the program might be:



How to

The program can be written as follows:

```
namespace Exam15
{
   class Program
   {
      static void Main(string[] args)
      {
         double sum = 0;
         int count = 0;
         double min = double.MaxValue;
         double max = double.MinValue;
         double number;
         while ((number = Enter()) != 0)
         {
            sum += number;
            ++count;
            if (min > number) min = number;
            if (max < number) max = number;
         }
         Print(count, sum, min, max);
      }

      static double Enter()
      {
         Console.Write("Enter a number (enter 0 to stop): ");
         string line = Console.ReadLine();
         return Convert.ToDouble(line);
      }

      static void Print(int count, double sum, double min, double max)
      {
         if (count > 0)
         {
            Console.WriteLine("Number of entered numbers: {0, 10:D}", count);
            Console.WriteLine("The sum of the numbers: {0, 10:F}", sum);
            Console.WriteLine("The smallest number: {0, 10:F}", min);
            Console.WriteLine("The highest number: {0, 10:F}", max);
            Console.WriteLine("Average: {0, 10:F}", sum / count);
         }
         else
            Console.WriteLine("There are no numbers entered...");
      }
   }
}
```


Download free eBooks at bookboon.com

Explanation

The method starts to declare and initialize some variables. The type *double* has two constants, and *MaxValue* and *MinValue* that is respectively the largest and the smallest occurring numbers of the type *double*. Note that this means that the variable *min* is initialized with a value that is too large, while the variable *max* is initialized with the value that is too small. This is followed by a while loop. Here you must particularly note the condition:

```
while ((number = Enter()) != 0)
```

Enter() is the same input method, which I have previously looked at. It returns a *double*, which is stored in the variable *number*. The value of the assignment is the value of the variable *number* after the method *Enter()* is executed, and it is this value that is compared with 0. Note that it is necessary with parentheses around the assignment, as the comparison operator != has higher precedence than the assignment. The result is that the *while* loop is repeated until the user enter 0.

Inside the while loop is what happens:

- the entered number is added to the *sum*
- the counter *count* is incremented by 1
- if the entered number is a new smallest number (less than that which before was the smallest), the variable *min* is changed
- if the entered number is a new largest numbers (greater than that which before was the largest), the variable *max* is changed

After the loop is completed, the *Print()* method are called to print the result. Here it is tested whether the variable *count* is greater than 0 to ensure that the program will not divide by 0.

Download free eBooks at bookboon.com

6 Strings

A *string* is a type that represents text, and it is an alias for the *System.String class*. A *string* is in contrast to the other types which have been treated so far, a reference type, but it is explained in detail in a later section. In C# a string contains Unicode characters, that is characters of the type *char*. Strings are important in practical programming, and you can't really imagine a program where you do not use strings. A string literal is specified as a sequence of characters in double quotation marks, for example:

```
string navn = "Volmer";
```

Often you need to specify special characters in a string. This is done by means of escape sequences, for example:

```
string navn = "\"Volmer\"";
```

if you want to specify the text "Volmer" incl. quotes. There are following escape sequences:

Click on the ad to read more

Download free eBooks at bookboon.com

Escape-sekvens	Karakter	Unicode kode
\'	Single quotation marks	0x0027
\"	Double quotation marks	0x0022
\\	Backslash	0x005C
\0	Zero	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Formfeed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

Variables of the type *string* generally do not result in major problems as they – although it is a reference type – are widely used as variables of the other simple types. However, one should be aware of statements of the form:

```
string str1 = null;
string str2 = "";
```

where the first is a *null* reference, that is a *string* that does not refer to anything, while the second refers to the empty *string*, a *string* with no content.

If you have a *string* object as

```
string text = "Volmer";
```

one can refer to individual characters with a 0-based index. If, for example you write

```
char ch = text[3];
```

the variable *ch* will have the value 'm'.

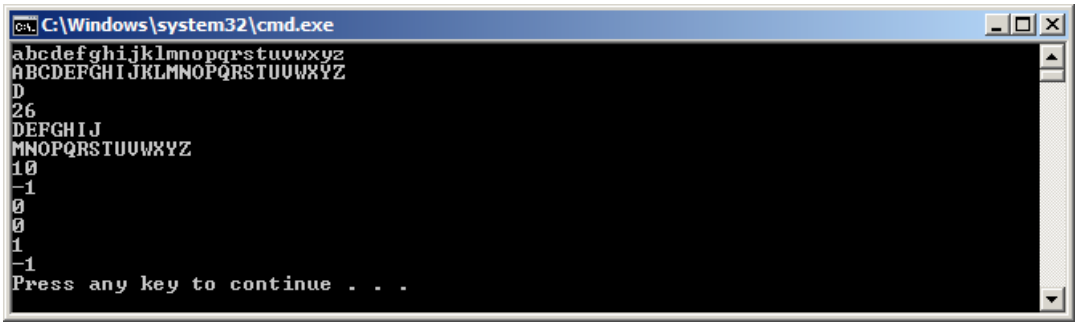
Exam16

The class string

The *class string* has a number of methods that are useful for the treatment of the strings. The following program provides examples of some of these methods:

Download free eBooks at bookboon.com

```
static void Main(string[] args)
{
   string text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   Console.WriteLine(text.ToLower());
   Console.WriteLine(text);
   Console.WriteLine(text[3]);
   Console.WriteLine(text.Length);
   Console.WriteLine(text.Substring(3, 7));
   Console.WriteLine(text.Substring(12));
   Console.WriteLine(text.IndexOf("KLMN"));
   Console.WriteLine(text.IndexOf("KMLN"));
   Console.WriteLine(text.Substring(3, 7).CompareTo("DEFGHIJ"));
   Console.WriteLine(
      String.Compare(text, 9, "1234567891234jklmnopq56789", 13, 8, true));
   string text1 = "Peter";
   string text2 = "Anders";
   Console.WriteLine(String.Compare(text1, text2));
   Console.WriteLine(String.Compare(text2, text1));
}
```



Explanation

First, initialize a variable *text*. The method *text.ToLower()* returns a string where all letters in *text* is converted to lowercases. *Length* is a property that returns the number of characters in a *string*. *Substring()* is a method that returns a substring. For example returns *text.Substring(3, 7)* a substring of 7 characters from position 3 onwards, that is *"DEFGHIJ"*. If you only specify one argument to *Substring()* the method returns a substring starting from that position until the end of the string. *IndexOf()* is a method that return the start position of the first occurrence of a partial *string*. For example returns *text.Indexof ("KLMN")* the value 10. If the substring is not found, it returns -1. *CompareTo()* is a method that compares the strings. If, you for example write

```
text.CompareTo("xyz")
```

it returns the value 0 if the two strings are equal, -1 if *text* is less than "xyz" and otherwise 1. There is also a *static* version for comparison of strings. For example will *String.Compare(text1, text2)* return 1, since the *text1* is greater than *text2*.

There are other methods for treatment of the strings, but the methods that are shown in this example, are the most important.

Download free eBooks at bookboon.com

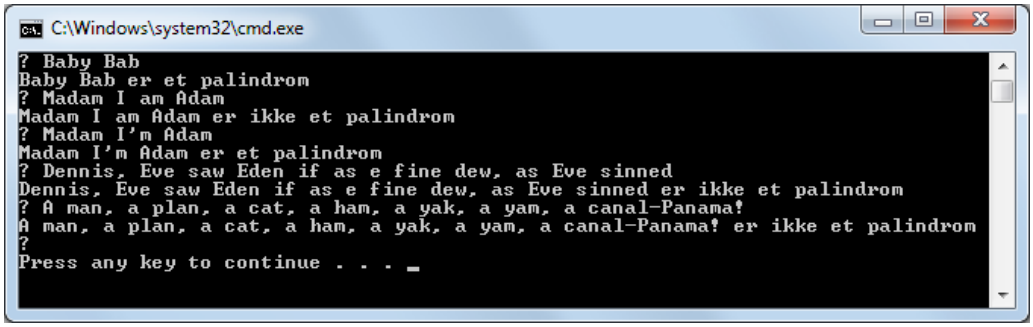
Exam17

Palindrome

As another example of a program about strings are the following a program which tests whether a string is a palindrome. A palindrome is a phrase that is spelled in the same way front and back, however, spaces and special characters are ignored, and there is no distinction between uppercase and lowercase letters. Examples of the palindromes are

- Baby Bab
- Dennis, Eve saw Eden if as a fine dew, as Eve sinned
- Madam, I'm Adam
- A man, a plan, a cat, a ham, a yak, a yam, a hat, a canal-Panama!

The program must act in the sense that the user inputs a sentence, then the program prints, whether it is a palindrome.



The example will partly show about characters and strings, but will also demonstrate several applications of control statements and conditions.

How to

The program can be described in the following manner:

1. enter a string
2. test whether it is a palindrome
3. print the result
4. repeat the above until the user keys Enter

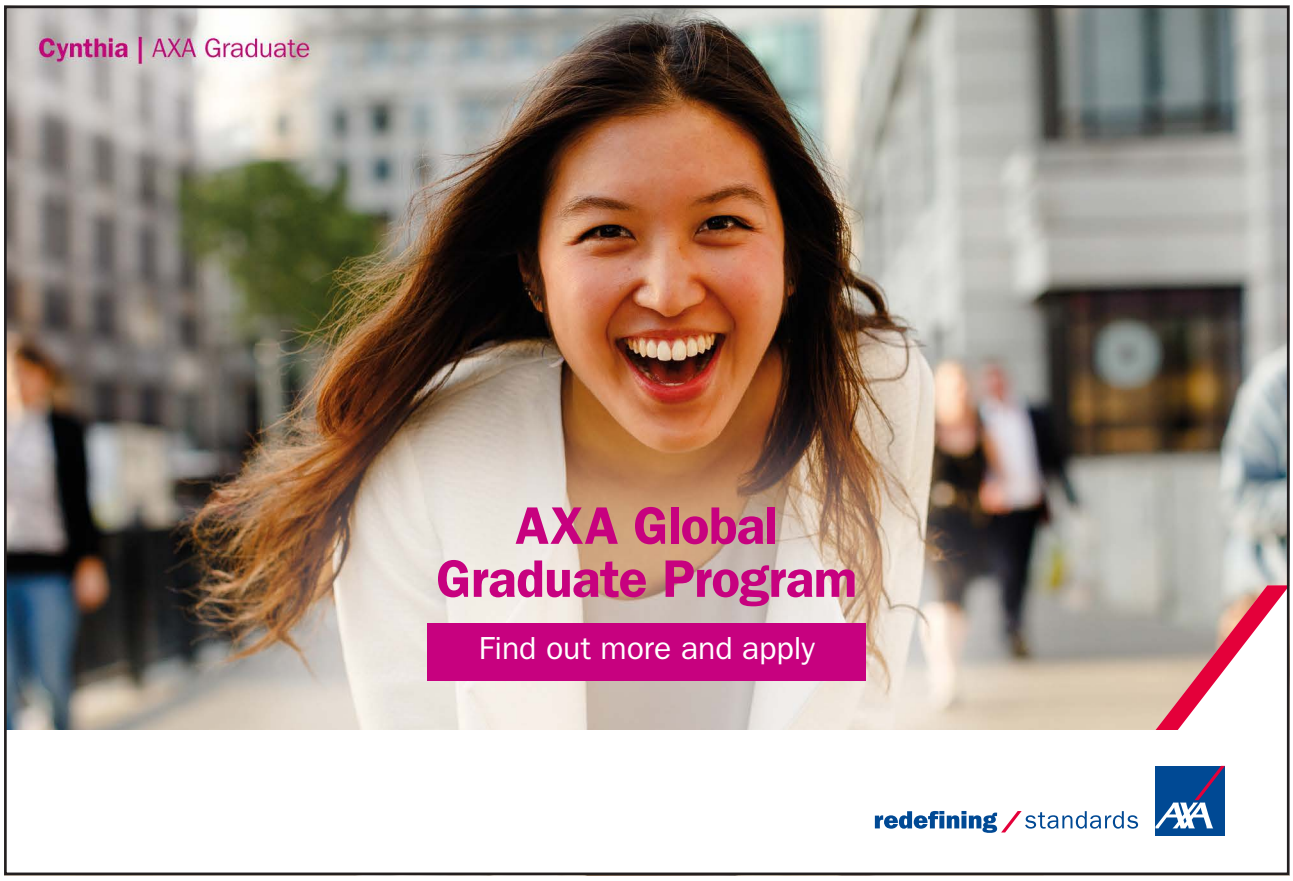
Entering is a simple task, as I have seen on several times. Print the result does not require much, and slightly more formal the solution can be formulated as follows:

```
repeat enter a line as long as the line is not empty
{
   if the line is a palindrome then
      print line is a palindrome
   else
      print line is not a palindrome
}
```


Download free eBooks at bookboon.com

Here is the biggest challenge to test if your line is a palindrome:

```
let i = 0
let j = index of the last character
as long as i < j repeat
{
   if both i and j refers to a character who must be ignored then
   {
      increment i by 1
      decrement j by 1
   }
   else if i reference to a character to be ignored then
      increment i by 1
   else if j reference to a character to be ignored then
      decrement j by 1
   else if character number i is different from character number j then
      it is not a palidrome
   ellers
   {
      increment i by 1
      decrement j by 1
   }
}
it is a palidrome
```



Click on the ad to read more

Download free eBooks at bookboon.com

This algorithm can be written in C# as follows:

```
static bool IsPalindrom(string text)
{
   for (int i = 0, j = text.Length - 1; i < j; )
   {
      if (!CharOk(text[i]) && !CharOk(text[j]))
      {
         ++i;
         --j;
      }
      else if (!CharOk(text[i]))
         ++i;
      else if (!CharOk(text[j]))
         --j;
      else if (text[i] != text[j])
         return false;
      else
      {
         ++i;
         --j;
      }
   }
   return true;
}

static bool CharOk(char ch)
{
   return (ch >= 'a' && ch <= 'z') || (ch >= '0' && ch <= '9') || ch == 'æ' ||
         ch == 'ø' || ch == 'å';
}
```

Then you can write the *Main()* method:

```
static void Main()
{
   while (true)
   {
      string line = Enter();
      if (IsPalindrom(line.ToLower())) Console.WriteLine(line + " er et palindrom");
      else Console.WriteLine(line + " er ikke et palindrom");
   }
}
```

Back is the input function:

```
static string Enter()
{
   Console.Write("? ");
   string text = Console.ReadLine();
   if (text.Length == 0) Environment.Exit(0);
   return text;
}
```


Download free eBooks at bookboon.com

Explanation

First, notice the condition in the while loop in *Main()*. It is always true, and it is thus an infinite loop. This means that the loop and thus the program must stop in some other way. It occurs in the input method. If the user simply keys enter and thus do not enter anything, the statement is executed:

```
Environment.Exit(0);
```

which stops the program. One can discuss whether it is the way to do it, and it is also primarily included here to demonstrate that it is possible.

The check for palindrome is performed by the method *IsPalindrom()*. The idea is to have an index to both the start and the end of the string. The characters are compared in pairs, and an auxiliary method *CharOk()* is used to test whether it is a character that must be included.

In the case of conditions, there are three major operators, which as arguments have one or two conditions:

- && takes two arguments of type *bool* and is a mathematical conjunction. This means that it is true if both its arguments are true.
- | | takes two arguments of type *bool* and is a mathematical disjunction. This means that it is true, if at least one of its arguments is true.
- ! has an argument of type *bool* and is a mathematical negation. That is, it has the opposite truth value of the argument.

The method *CharOk()* is used to test whether its parameter is a character to be counted: Whether it is a letter or a digit. The characters are arranged in natural order, but the three Danish letters are for themselves, which complicates the control a bit. You will notice how the complex expression is built with the operators && and | |. && has higher precedence than | |, so strictly speaking, the parentheses can be omitted, but they increase readability. The check may be expressed in text something like:

```
if (ch is at least a and ch is lower than z) or
   (ch at least 0 and ch is lower than 9) or
    ch is æ or ch is ø or ch is å then ...
```

The method *IsPalindrom()* use the ! operator to test the opposite value of the *CharOk()*, for example:

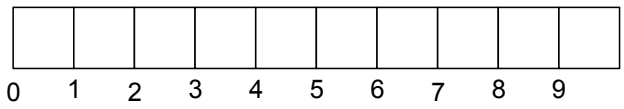
```
if not CharOk(text[i]) then ...
```

Note that when the method *IsPalindrom()* is called in the *Main()* method the text (the string entered) are first converted to lowercase with the method *ToLower()*. It is therefore, it is sufficient to test for the small letters in the method *CharOk()*.

Download free eBooks at bookboon.com

7 Arrays

An array is a number of objects of a particular type that can be referenced with a common name and the individual objects are referenced by that name and an index. The first element always has index 0, and you can have the following picture of an array with 10 objects



where each box has space for an object of that type. In C# you can create an array with space for 10 *int* objects as follows:

```
int[] number = new int[10];
```

It is the brackets [] which says that *number* is an array. Notice how an array is created with *new*. It is necessary and explains that an array is a reference type. If, for example you will place numbers in the first four places, it could be done as follows:

```
number[0] = 2;
number[1] = 3;
number[2] = 5;
number[3] = 7;
```

The following is also a legal statement:

```
int sum = number[1] + number[3];
```

and the variable *sum* will have the value 10. A single element like *number[2]* can simply be used anywhere where one can use an *int* variable.

Exam18

Two arrays of the type *int*

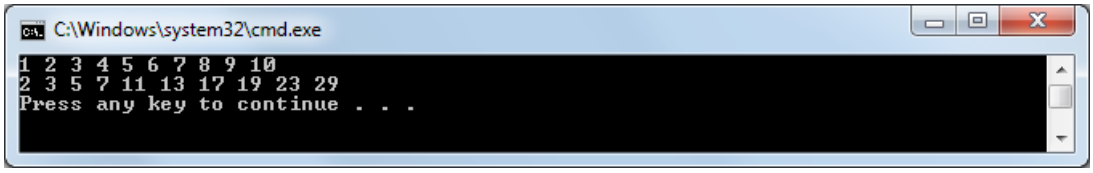
As an example, the following program creates two arrays with elements of the type *int* and prints them on the screen:

```
static void Main(string[] args)
{
   int[] v1 = new int[10];
   int[] v2 = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
   for (int i = 0; i < v1.Length; ++i) v1[i] = i + 1;
   Print(v1);
   Print(v2);
}
```


Download free eBooks at bookboon.com

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

If you run the program, you get the result:



Explanation

You must note how the array *v2* is defined by a list of numbers. It simply means that the compiler can create the array and directly initialize it with the list elements. Also, note how the array *v1* is initialized in a *for* loop. Note here especially how *Length* can be used to refer to the number of elements. Note, finally, how to transfer an array as a parameter to a method, and specifically how you in the method *Print()* write that *t* is an array.

Click on the ad to read more

Download free eBooks at bookboon.com

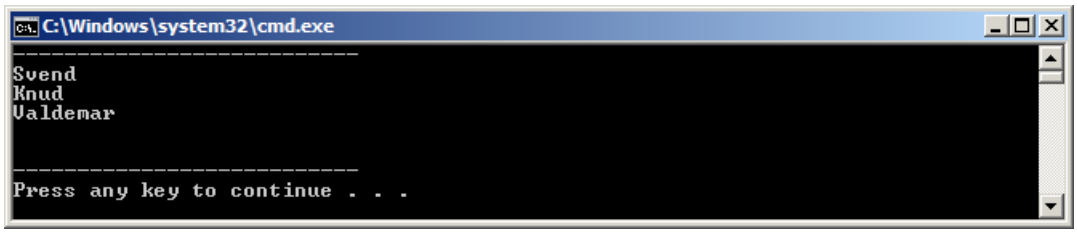
Exam19

Array of strings

In the above example, the type of the arrays is *int*, but the type of an array can generally be anything. Below is a program that creates an array with 5 objects of the type *string*:

```
static void Main(string[] args)
{
   string[] names = new string[5];
   names[0] = "Svend";
   names[1] = "Knud";
   names[2] = "Valdemar";
   Console.WriteLine("---------------------------");
   for (int i = 0; i < names.Length; ++i) Console.WriteLine(names[i]);
   Console.WriteLine("---------------------------");
}
```

Note that only the first three elements are initialized. The latter two are not initialized and has the value *null*, equivalent to that string is a reference type. If the method is executed, one gets the result:



Note the two blank lines corresponding to the printing of null.

Exam20

Yatzy

There should be written a program to simulate where you throw 5 cubes until they all show the same value – until you get yatzy.

How to

The program can be written as follows:

```
class Program
{
   static Random rand = new Random();

   static void Main(string[] args)
   {
      int[] beaker = new int[5];
      int count = 0;
      do
      {
```


Download free eBooks at bookboon.com

```
            Cast(beaker);
            Show(beaker);
            ++count;
        }
        while (!Equals(beaker));
        Console.WriteLine("You've got yatzy after {0} attempts", count);
    }

    static void Cast(int[] beaker)
    {
        for (int i = 0; i < beaker.Length; ++i) beaker[i] = rand.Next(1, 7);
    }

    static void Show(int[] beaker)
    {
        Console.Write("[");
        for (int i = 0; i < beaker.Length; ++i) Console.Write(" {0}", beaker[i]);
        Console.WriteLine(" ]");
    }

    static bool Equals(int[] beaker)
    {
        for (int i = 1; i < beaker.Length; ++i)
            if (beaker[i] != beaker[0]) return false;
        return true;
    }
}
```

Explanation

In order to simulate the game is a need for a random number generator. .NET has a class *Random* to that purpose and the program starts therefore, to create a random number generator called *rand*. Note that it is created outside of all methods. This means that there is access to it anywhere in the program, and hence that all methods know it.

The program will simulate a dice cup with 5 dice (cubes). In *Main()* is defined an array of 5 *int* elements to simulate the cup. There is also defined a variable *count*, which will be used to count how many times you have to toss the cup until you get yatzy.

In addition to *Main()* there are three methods. The first simulates that you toss with the cup. Here especially note how the random number generator is used: *rand.Next (1, 7)* means that one will have a random number which is greater than or equal to 1 and less than 7, and thus a random of the numbers 1, 2 , 3, 4, 5 and 6. The method *Show()* does nothing more than to the print the contents of the cup on the screen.

The method *Equals()* is used to test whether the 5 dice in the cup are the same. Here you should note two things: The algorithm and the return value. Regarding the latter is the methods return value something discussed in a later section, but a method that has a value, can for example be used in a condition as here in the *Main()* method. Note also how the method's value is assigned with a *return* statement. The algorithm consists in comparing all the cubes from index 1 and up to the first cube – cube with index 0. If you find one that differs from the first, all the cubes are not identical, and the method returns *false*. If in contrast you go throughout the loop, all the dice are the same, and the method returns *true*.

Download free eBooks at bookboon.com

Exam21

Craps

This example show a program that simulates a simple dice game called craps:

A player throws two dice, 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, you record the sum as the player's points. The player then proceeds to throw the dice until the sum of the eyes either is the player points or until the sum is 7. Is the sum are the player's points the player has won. Is the sum is 7, the player has lost, and the house wins.

The program will operate in the way that the user must first enter how many games you want to play. Then the program simulates the desired number of games and finally prints, how many times a player has won and how many times the house has won.

How to

This time, the solution is not quite simple, and there are several options. I will try with a kind of decomposition of the task into smaller parts, and the *Main()* method can be written as follows:

Click on the ad to read more

Download free eBooks at bookboon.com

```
Main()
{
   let count = the number of games
   repeat count times
   {
      Play
      if player has won then add 1 to the variable won
      print who is the winder
   }
   print the result
}
```

From this sketch of the *Main()* method, it is clear that the most important is Play(), which is a method that simulates a single game, and thus the method to implement the rules for craps. The algorithm can be written as follows:

```
Play()
{
   bg is a cup with two dice
   toss with bg and notes the sum of the dice eyes as points
   if points is 7 or points is 11 then the player has won
   else if points is 2 or points is 3 or points is 12 then the house has won
   else
   {
      toss with bg and note the sum of the eyes as sum
      as long as sum is different from the points and the sum varies from 7 repeat
      {
         toss with bg and note the sum of the eyes as sum
      }
   }
   if sum is equal to points then the player has won elllers the house has won
}
```

The algorithm for *Play()* basically consists of simple operations, except perhaps the toss with the cup that can be described as follows:

```
1.Roll the dice
2.Print the content of the cup
3.Determine the sum of the eyes of the dice
```

With everything in place, there essentially only remains to translate the above algorithms to C#, and I will start with the latter and thus to simulate how you toss with the cup:

```
static int Throw(int[] beaker)
{
   Cast(beaker);
   Show(beaker);
   return Sum(beaker);
}

static void Cast(int[] beaker)
{
   for (int i = 0; i < beaker.Length; ++i) beaker[i] = rand.Next(1, 7);
}
```


Download free eBooks at bookboon.com

```
static void Show(int[] beaker)
{
   Console.Write("[");
   for (int i = 0; i < beaker.Length; ++i) Console.Write(" {0}", beaker[i]);
   Console.WriteLine(" ]");
}

static int Sum(int[] beaker)
{
   int sum = 0;
   for (int i = 0; i < beaker.Length; ++i) sum += beaker[i];
   return sum;
}
```

where *rand* is a random number generator. As shown above, it consists of three operations:

1. Roll the dice is implemented as the method *Cast()*
2. Print the content of the cup is implemented as the method *Show()*
3. Determine the sum of the eyes of the dice and return the value is implemented as the method *Sum()*

Note that the three methods are relatively simple and all are written so they are independent of the number of cubes in the cup.

Using this method *Throw()*, you can write the algorithm for *Play()* in C#:

```
static bool Play()
{
   int[] beaker = new int[2];
   int point = Throw(beaker);
   if (point == 7 || point == 11) return true;
   if (point == 2 || point == 3 || point == 12) return false;
   int sum = Throw(beaker);
   while (sum != point && sum != 7) sum = Throw(beaker);
   return sum == point;
}
```

Note that the method is essentially a direct copy of the above algorithm.

Next the *Main()* method:

```
static void Main()
{
   int count = Enter();
   int won = 0;
   for (int i = 0; i < count; ++i)
      if (Play())
      {
         Console.WriteLine("Player has won...");
         ++won;
      }
      else
         Console.WriteLine("The house has won...");
   Result(count, won);
}
```


Download free eBooks at bookboon.com

```
static int Enter()
{
   Console.Write("Enter the numbers of games: ");
   string text = Console.ReadLine();
   return Convert.ToInt32(text);
}
```

Explanation

I will not systematically review the entire code, but merely pointing out things that you should special to be aware of. The program does not contain anything new compared to what is shown in earlier examples.

Note therefore that several methods have a return value. Note for example the *Enter()* method that returns an *int*. Notice how the method's return value in *Main()* is stored in the variable *count*.

The method *Play()* performs a game. It creates a cup as an array with two cubes (a cube is again represented as an *int*). You throw the dice the first time and note what the dice show (variable *point*). Then test whether the player has won, or whether the house has won. If the player has won, the method returns *true*, and if the house has won the method returns *false*. If the game is not decided after the first roll, roll again and repeat until the game is decided.

Note also that the methods *Cast()* and *Show()* are direct copies of the corresponding methods from the previous program.

Comment

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 is almost the same probability of winning with a slight predominance for the house – and it should also like to be if the house should not go bankrupt.

Download free eBooks at bookboon.com

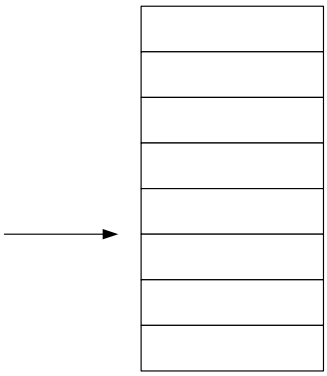
Part 2 Object Oriented Programming

Applications must process data, and data must be represented and stored, and that is what variables are used for. A language like C# has a number of built-in types for variables, but often would you need to define your own data types that better reflect the job that the program must solve. That is where the class concept comes into play. A class is something that defines a concept or thing within the program's area of concern, and there can be said a lot about what are classes and what not, but basically is a class a type. It's a bit more than just a simple type as a class partly defines how data should be represented, but also what you can do with the data of that type. A class defines both how the type should be represented and what operations you can perform on the variables of that type.

When you create variables whose type is a class, we talk about objects, so that a variable of a class type is called an object, but really there is no much difference between a variable and an object, and there is a good piece of the road no reason to distinguish between the two. But dig a little deeper, there is a reason that has to do with how the variables and objects are allocated in the machine's memory.

Click on the ad to read more
Download free eBooks at bookboon.com

Variables of types such as *int*, *double*, *bool*, *char*, etc. are called variables of simple types. For a running program is allocated a stack, that is a memory area used by the application to store variables. The stack is very effective so it is very effective to the program continuously setting up and removal of variables if necessary. This is known as a stack, because one can think of a stack as a data structure illustrated in the following manner:



When the program creates a new variable, it happens at the top of the stack – where the arrow points, and when a variable is removed, it is the one that lies at the top of the stack. All of the simple data types have the property that a variable always have the same size. For example fills an *int* the same regardless of what value it has. Therefore, such variables are stored directly on the stack, because the compiler knows how much they fill. If, for example you write as follows

```
int a = 23;
```

then there will on the stack be created a variable of the type *int* with the space that an *int* requires and the value 23 are stored there. Variables that are stored directly on the stack in this way are called *value types*, and all the built-in types – except *string* – are value types.

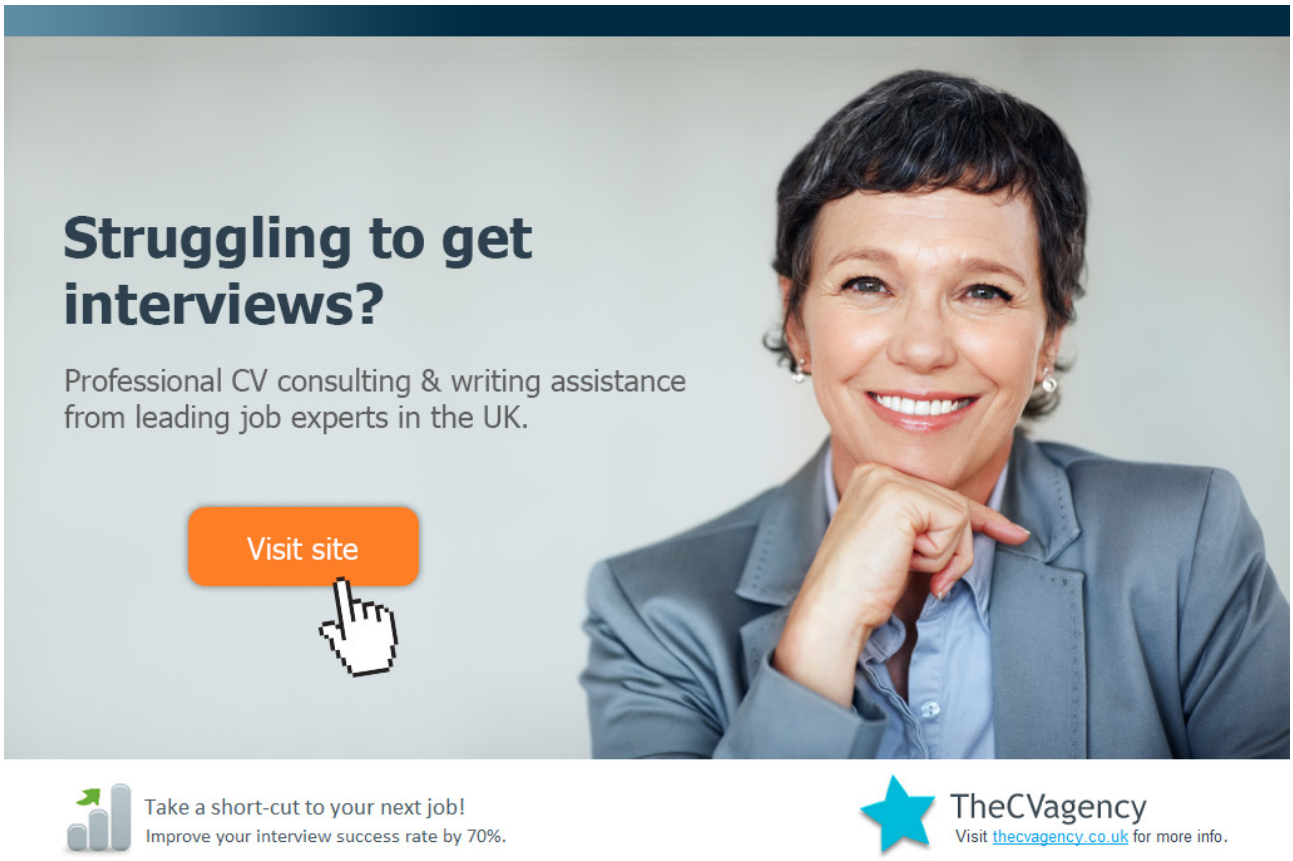
Things are different with variables of reference types such as variables that have a class type. They must be explicitly created with *new*. If, for example you have a class named *Coin* (see below) and you want to create such an object, you must write

```
Coin c = new Coin();
```

It looks like, how to create a simple variable. The variable is named *c*. When *new* are performed what happens is that on the so-called *heap* there are created an object on the basis of the type *Coin*. One can think of the heap as a memory pool from which the machine can allocate memory as needed. On the stack is created as usual, a variable with the type of *Coin*, but what it is saved on the stack, is not the value of the *Coin* object, but instead a reference to the object on the heap. All references have the same size regardless of the type and can therefore be stored on the stack. That's why we call it a reference type. How exactly one object is created on the heap is determined by a so-called *heap manager*, there is a program that constantly runs and manages the heap. It is also the heap manager, which automatically removes an object when there is no longer needed.

Download free eBooks at bookboon.com

For the foregoing reasons, it is clear that the variables of value types are more efficient than objects of reference types. It does not mean that objects of reference types are ineffective, and in most cases it is not a difference which one needs to take care of, but conversely, there are also situations where the difference matters. It is thus important to know that there is a big difference in how value types and reference types are handled by the system and that in some contexts it have a major impact on how the program will behave, but there'll be many examples which clarify the difference. So far it is enough to know that the data elements can be grouped into two categories depending on their data type, so that data of value types are allocated on the stack and usually called variables, while data of the reference types are allocated on the heap and are called objects – even if there is not complete consistency between the two names.



Download free eBooks at bookboon.com

8 Classes

After this rather technical explanation it is time to deal with classes in C#.

Exam22

Coins

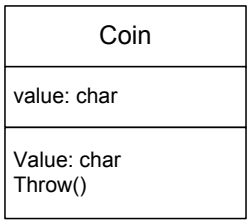
You must write a program to simulate that you throw two coins. The program will throw the coins until they both show crown (Head). If you run the program, the result could be the following:

How to

There is a need for a type which can represent a coin. In this case, the coin's value will be head or tail and it must provide the following services available:

- one should be able to throw the coin (simulating that it gets a random value)
- one should be able to read the coin's value (see what the coin shows)

The type is called *Coin* and may be illustrated as follows:



The code can be written as follows:

```
class Program
{
   static void Main(string[] args)
   {
```


Download free eBooks at bookboon.com

```
      Coin c1 = new Coin();
      Coin c2 = new Coin();
      do
      {
         c1.Throw();
         c2.Throw();
         Console.WriteLine(c1.Value + " " + c2.Value);
      }
      while (c1.Value != 'H' || c2.Value != 'H');
   }
}

public class Coin
{
   private static Random rand = new Random();
   private char value;

   public Coin()
   {
      Throw();
   }

   public char Value
   {
      get { return value; }
   }

   public void Throw()
   {
      value = (rand.Next(2) == 0) ? 'H' : 'T';
   }

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

Explanation

You should represent a concept that here is a coin, and you can do that with a *class*, here named *Coin*. Note first that the class *Coin* has no *Main()* method and the methods are not declared *static*. A class is a concept – a type – which may be used in a program.

In C# a class contains four elements:

- instance variables
- constructors
- properties
- methods

and all four elements are present in the above class. The class has two variables:

```
private static Random rand = new Random();
private char value;
```


Download free eBooks at bookboon.com

The first is called *rand* and the type is *Random*. It has been declared *static* and is a bit special, so it is explained shortly. The second is called *value* and has the type *char*. It is the member variable that represents the coin. Note that both variables are declared *private*. This means that there is only access to these variables from the class's own methods – they can't be directly referenced from other classes that use the class *Coin*.

The class has a method called *Throw()*. When the method is performed, it simulate that the coin is thrown, thus giving it a new value:

```
value = (rand.Next(2) == 0) ? 'H' : 'T';
```

The method assigns the variable *value* a new random value. To that purpose it use the ? operator. It is an operator which has the form

condition ? expression1 : expression2;

and the meaning is that if the condition is true, the value is the value of expression1 and else the value of expression2.

Click on the ad to read more

Download free eBooks at bookboon.com

Random is a class in the namespace *System*, and it has a method *Next()* that returns a random non negative integer. If you specify a parameter as *Next(n)*, you gets a random value between 0 and n-1, both incl. Therefore

```
rand.Next(2)
```

is a random value 0 or 1, and equivalent thereto assigned to the member variable *value* a random character H and T.

The member variable *value* is *private* and can only be used inside the class *Coin*. The program, that should use the *Coin* class, has to know the value of a coin that is read the variable value. For that the class *Coin* has a property:

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

The property has the same type and same name as a member variable – just you write the name in uppercase. Note that a property is not syntactically a method (there are no parentheses after the name). The property has a get part that returns the value of the current member variable.

When creating an object of class *Coin*, the machine also creates an instance variable *value*. The coin can have only two legal values (H, T), and when the coin's value is the value of the variable *value*, the coin will have an illegal value until it is thrown. It's unfortunate, because that coins in the real world will always have a legal value, but also because a program that uses *Coin* class would fail. This problem is solved with a *constructor*. Syntactic is a method that does not have any type and which has the name of the class:

```
public Coin()
{
   Throw();
}
```

A constructor is a method that is automatically executed when creating an object of a class, and it is typically used to initialize instance variables. In this case, the constructor throws the coin and thus ensures that a coin has a legal value once it is created. Note also that the only way a *Coin* can change its value is by throwing it. It is guaranteed by the member variable *value* is private.

When you have the class, you can create objects whose type is the *Coin*. You do it with the *new* operator, for example:

```
Coin c1 = new Coin();
```


Download free eBooks at bookboon.com

It creates an object named *c1* whose type is *Coin*. When the *new* operator is performed, then the *class's* non-static member variables are created, and the class's constructor is automatically performed. The static member variables are created in a different way, and they are created only once. This means that static members are shared between all objects of the class.

In this case the member variable *rand* is static and all objects of the class *Coin* will therefore apply the same *rand* variable. It is important, because otherwise all *Coin* objects initialize their own random number generator (which is initialized by reading the hardware clock) and the result would be that two objects created in the same place in the program would always have the same values.

In *Main()* the program creates two objects, called *c1* and *c2*. After they are created the program goes into a *do* loop that throws the coins, and displays them on the screen. The loop runs until both coins show the value H. Notice how you throw a coin:

```
c1.Throw();
```

c1 is an object, and *Throw()* is a method in the class that defines the object. You execute the method on the current object using the dot operator. Note also how to refer to the coin's value:

```
c1.Value
```

Here you use the class's property, which returns the coin's value. Note that in this case, the class *Coin* has a property which is read only. Classes may also have properties that are read write, which will be apparent from other examples.

The class has also a method which is called *ToString()*, and which are not used in this example. Classes should generally have a *ToString()* method that returns a string that is a text representation of a concrete object. This means that an object can be printed with the method *System.WriteLine()*. Note that the method is defined as *override*. I will explained the meaning later.

Comment

A class has a *type of visibility* that is either *public* or *internal*. The class *Coin* has *public* visibility. This means that anyone can create objects of this type. Visibility can also be specified as *internal*:

```
internal class Coin
```

Such a class can be instantiated from classes in the same assembly (dll or exe) as the class itself. If you do not set visibility for a class the default is *internal*. Class members also have a visibility that is one of the following:

Download free eBooks at bookboon.com

- *public*
- *private*
- *internal*
- *protected*
- *protected internal*

Preliminary I will mention only the first three. *public* means that a member can be referenced by methods in other classes, while *private* means that a member can only be referenced from the class's own methods. *internal* means that a member can be referenced by all methods in the same assembly. Members visibility can not override the type visibility. If, for example a class has *internal* visibility, you can not reference a public method from another assembly.

Both variables and methods can be assigned visibility, but usually instance variables are defined *private* (or *protected*). It is a principle often called data encapsulation and ensure that it is the one who writes the class that determines what access the outside world, have to the class's instance variables. There is only access to instance variables through the class's methods and properties, and thus through the services that the class provides. If, for example you look at the class *Coin* above, the variable *value* are only changed in the method *Throw()*, and hence we can be sure that a *Coin* always have a legal value. If the value was published, a user could write some thing like

```
Coin c = new Coin();
c.value = 'X';
```

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

That would mean both that the object now has illegal value, but what is worse, such an error can be very difficult to find. Another reason to always declare a instance variable as *private* is that a variable in this way, is an internal characteristic of the class, as users of the class, in principle, does not know. In the class *Coin* has variable *value* the type *char*. If for one reason or another you want to change this type to a *string*:

```
public class Coin
{
   private string value;
```

you can do it without affecting the applications that use the class. There are no directly refer to the variable value.

The conclusion is that it is a principle OOP to define instance variables *private* (or *protected*).

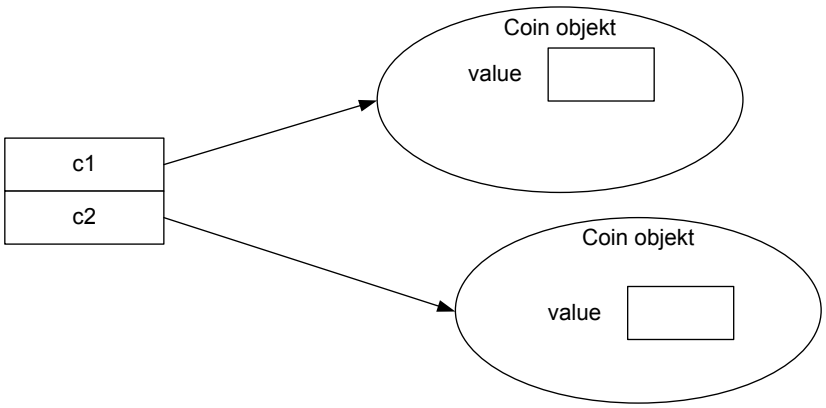
Comment

When you create an object of a class, the class's instance variables are created, and then the class's constructor are executed. If a class has no constructor, a default constructor will automatically be created, that is a constructor without parameters. A constructor is characterized by the fact that it is a method which has the same name of the class and do not have any type. A constructor is a method, but it can't be called explicitly and are executed only when an object are instantiated. The *Coin* class has a constructor, a default constructor, but a class may well have several constructors, as applicable to the general rules for overriding the methods.

Coin is a reference type, and the following statements create two objects:

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

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

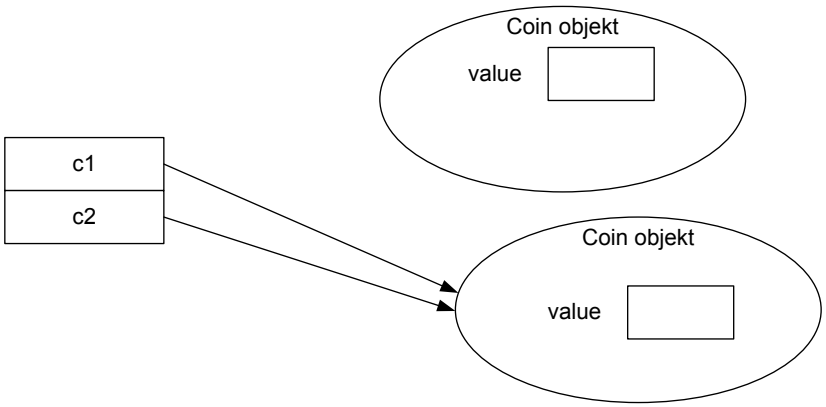


Download free eBooks at bookboon.com

If, for example you writes

```
c1 = c2;
```

this means that there is no longer any reference to the object before referring to as *c1*, but there are two references to the object referring to by *c2*.



If then you writes

```
c1.Throw();
c2.Throw();
```

this means that the you throw the same coin twice, because both references refer to the same object.

When there no longer are any references to an object, it also means that the object is automatically removed of the heap manager and the memory that the object has used, is released.

Download free eBooks at bookboon.com

9 Design of classes

A class is a type, but it is also a design concept. The class defines the objects in the form of instance variables, how objects are created and what memory that is allocated to objects. Objects have at any given time a value in form of the instance variables content and an object's value is usually referred to as its state. The class also defines in terms of its methods, what you can do with the object, and thus how to read and modify the object's state.

Every C# program consists of a family of classes that together define how the program would operate and a running program consists of a number of objects instantiated in the context of the programs classes, objects that work together to accomplish the thing which the program must do. The work of writing a program is thus to write the classes that the program will consist of. Which classes are, on the other hand not very clear, and the same program can typically be written in many ways built up of classes that are completely different. 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 gets the job done correctly, it may be irrelevant, which classes it consists of, but inappropriate classes means


Click on the ad to read more

Download free eBooks at bookboon.com

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

Therefore are design and choice of classes a very central issue in the context of programming and one are speaking of program quality (or lack of the same). A class is more than just a type, but it is a definition or description of a concept within the program area of concern. When you have to define which classes the program will consist of, one must therefore largely be focusing on classes as a tool to define the essential concepts more than on classes as a type in a programming language.

An object is characterized by four things:

- a unique identifier that identifies a particular object in relation to 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 where the object are created to it again are deleted

An object is created from a class, and assigned at that time 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 state of the object. The class's methods define what can be done with the object and hence the object's behavior. The last point of concerning is object's life cycle linked to the concept of scope, as explained later.

Exam23

Dice

The task is to write a program that simulates that you throw 5 cubes until they are all alike. Finally the program will print out how many times the cubes is thrown – an example which, incidentally, I've seen in the past (Exam20), but by that time I had no concept of class available.

How to

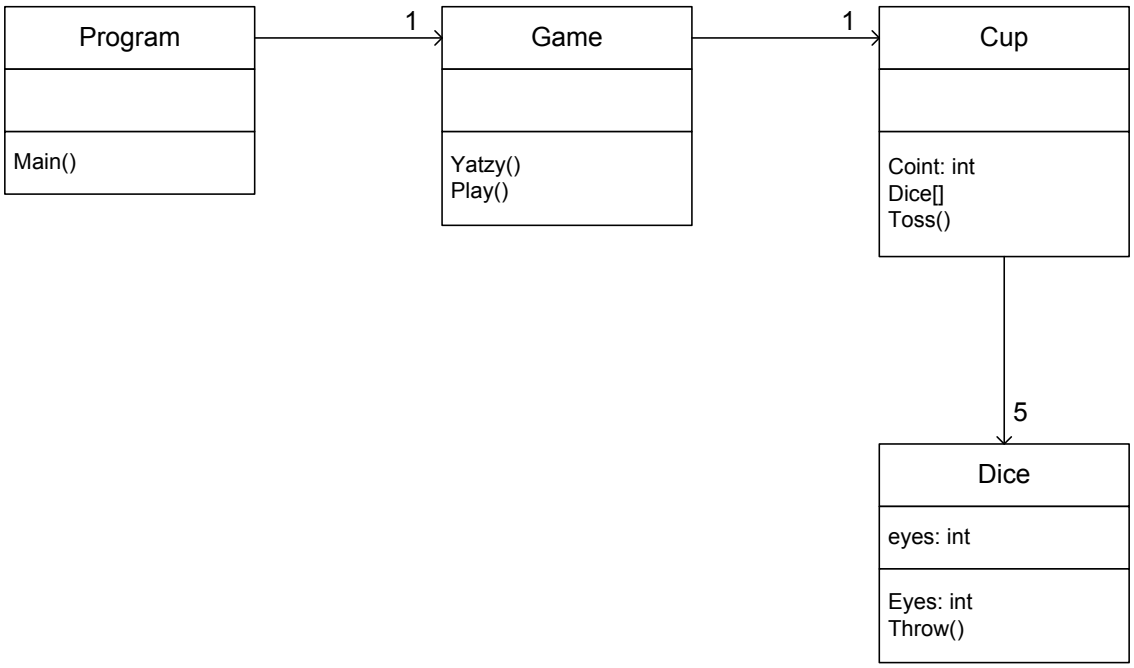
The first task is to find (decide) the program's classes, and basically we need a class that can represent a die (a cube), where a die is characterized by a number of eyes, and it should be possible to throw the die and get knowing what it shows.

Another obvious class is a cup with 5 dice, and here it must be possible to toss the cup and get to know what the dice show.

Download free eBooks at bookboon.com

Finally, I will write a class that can represent the actual game. That is, a class which has a cup and a method which carries out the game.

This corresponds to the following design:



Note that the design is somehow arbitrarily in the sense that I have not followed certain guidelines, but the choice of classes is made by the developer, and there could be other solutions that could be just as good. Design is largely an activity, based on experience and by watching and relate to what others have done, and it is seldom possible to decide exactly what is the best design but a design can be more or less appropriate.

It should be added that there are methods and techniques that can help developers to find a program's classes. These are just issues which fall outside the scope of this book, but for larger projects are that kind of systems development methodologies not only useful but also necessary.

If I were to attempt a justification of the above design, then the task is to write a program that can simulate a simple dice game. The term I first catch sight of, is a die, as something of what all the fuss is about. Therefore a class *Dice*, which may represent a die. In the game, you could throw 5 dice and inspired by the way, you for example plays Yatzy, it is natural to think of a cup, which you can toss with. Therefore I have a class *Cup*. Finally the game rules and logic to play must be somewhere and it is not properties of a cup, because in the real world one does not produce cups for certain games – a cup is general and can be applied to many kinds of games. Then the game itself is represented by a class *Game*.

With this design in place, it is relatively simple to write each class. The class of a die can be written as follows:

Download free eBooks at bookboon.com

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

   public Dice()
   {
      Throw();
   }

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

   public int Eyes
   {
      get { return eyes; }
   }

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

This e-book
is made with
SetaPDF

SETASIGN

PDF components for PHP developers

www.setasign.com

Click on the ad to read more

Download free eBooks at bookboon.com

Then there is the class for a cup that can best be characterized as an encapsulation of an array with *Dice* objects:

```
public class Cup
{
   private Dice[] dice;

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

   public int Count
   {
      get { return dice.Length; }
   }

   public Dice this[int i]
   {
      get { return dice[i]; }
   }

   public virtual void Toss()
   {
      for (int i = 0; i < dice.Length; ++i) dice[i].Throw();
   }

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

The class to the game itself is very simple and is basically a question that the code from Exam20 has moved into its own class:

```
public class Game
{
   private Cup cup;

   public Game(int n)
   {
      cup = new Cup(n);
   }

   public void Play()
   {
      int count = 0;
      do
      {
         cup.Toss();
         ++count;
         Console.WriteLine(cup);
      }
```


Download free eBooks at bookboon.com

```
      while (!Yatzy());
      Console.WriteLine("You've got yatzy after {0} attempts", count);
   }

   private bool Yatzy()
   {
      for (int i = 1; i < cup.Count; ++i)
         if (cup[i].Eyes != cup[0].Eyes) return false;
      return true;
   }
}
```

Finally, there is the *Main()* method:

```
class Program
{
   static void Main(string[] args)
   {
      Game game = new Game(5);
      game.Play();
   }
}
```

Explanation

If you compare the class *Dice* with the class *Coin* you will note that in principle they are similar and there is not much to explain except that the data representation this time is an *int*.

The class *Cup* represents a dice cup, which is implemented as a class with a container for *Dice* objects. The container is an array:

```
private Dice []dice;
```

but to make the cup more flexible and allow it to be used in other contexts (where it does not necessarily have to contain 5 cubes) the number of cubes is a parameter to the constructor. *Dice* is a reference type, and when you create an array of reference types, you get an array that contains references, but not objects of that type. They must be created explicit:

```
for (int i = 0; i < n; ++i) dice[i] = new Dice();
```

Since the class has a constructor with a parameter, it has no default constructor. It means that you can't create a cup in the following way:

```
Cup cup = new Cup(); // illegal
```

The class has a read only property that returns the number of cubes in the cup. It is necessary for users of the class. Otherwise they can't get to know how many cubes the cup contains.

In addition, the class has a read only property that returns the cube that has index *i*:

```
public Dice this[int i]
{
   get { return dice[i]; }
}
```


Download free eBooks at bookboon.com

This is a somewhat special syntax that uses *this* as a property name, and I will not go into detail here, but it means that in a program you can write the following:

```
Cup cup = new Cup(5);
Dice d = cup[1];
```

and refer to the individual cubes via an index – that is as if a cup was an array. The user can then use the class as if it were an array, but the user can't see (or need not have knowledge about) whether the internal representation in the class is an array or something else. In this case, these are a read only property. The fact that it is read only, means that a user for example can't write:

```
cup[1] = new Dice(); // ulovligt
```

The user can not put another cube in the cup, but since the operator returns a reference the user can do anything with the cubes in the cup, for example

```
Dice d = cup[1];
d.Throw();
```

The method *Toss()* is simple and consists merely of a loop that runs over all cubes and throwing them.

With the class *Cup* is the class *Game* simple and it has a constructor with a parameter, and based on that parameter it creates a *Cup* with the number of cubes that you want to play with. The game itself is carried out by the method *Play()*.

Note here especially the method *Yatzy()* that tests that all cubes are the same. It is a private method, since it only should be used in class *Game*.

Comment

The class *Dice* has a property that tells what the cube show:

```
public int Eyes
{
   get { return eyes; }
}
```

Technically speaking, this means that there automatically is created the following method:

```
public int get_Eyes()
{
   return eyes;
}
```

A property is thus basically the same as a method but with a different syntax. The goal of a property is that the user uses a property as it was a variable, but still with the security that it is the programmer of the class that determines what is possible. Above it is thus the programmer of the class *Dice*, who has determined that the user must read a cube eyes, but not change the value.

Download free eBooks at bookboon.com

10 Methods

This section elaborates on issues relating class methods, and it is primarily about four things:

- methods names
- methods return value
- properties
- methods parameters

and here it is about methods parameters which are most to say.

Methods names

About methods names there are not much to say beyond that is the same as names for variables. However, it has been a convention in C# that the name of a method always starts with an uppercase letter. A method is identified by its name and its parameters. A class may well have multiple methods with the same name as long as they have different parameters, either in terms of number or types.

Click on the ad to read more

Download free eBooks at bookboon.com

Exam24

Function overriding

This example shows a program and a class with three methods of the same name.

How to

The class *Program* has the following methods, all with the same name:

```
static int Max(int a, int b)
{
   return a < b ? b : a;
}

static double Max(double a, double b)
{
   return a < b ? b : a;
}

static int Max(int a, int b, int c)
{
   return Max(Max(a, b), Max(b, c));
}
```

Explanation

The compiler may separate the two first, when the two parameters that are of different type, and may also separate the last when it has three parameters.

Notice how the ? operator is used. Also note how the last method actually calls the first.

Note that methods can't be separated on return type, but only on the parameters.

The fact that a method can't be identified by the name only, but also of its parameters, are sometimes called function overloading. Note also that it is not something that is special to methods, but also applies to operators. If, for example you write

```
c = a + b;
```

it means something different depending on which types of variables *a* and *b* are – for instance if the type is an *int* or a *string*.

Comment

In this example I have used the question operator. I have previously discussed this operator, but since it is an operator, which some feel is hard to understand, it must have an additional comment here. In a program you can write something like the following:

```
if (a < b)
   m = a;
else
   m = b;
```


Download free eBooks at bookboon.com

that assigns the variable *m* the smallest of the variables *a* and *b*. That means that if the condition a < b is true, given *m* value of *a*, otherwise the value of *b*. You can also write the same with the question operator

```
m = a < b ? a : b;
```

and the meaning is exactly the same. The question operator starts with a condition and if it is *true*, the value of the expression is the value of the expression (variable) after the question mark and if not the value of the expression is the value after the colon. In this example the statement

```
return a < b ? b : a;
```

will return the largest of the variables *a* and *b*. The question operator is often an alternative to an *if* statement, but it is of course the difference that it is an operator and thus can be included in expression as in the above return statement. As mentioned, there are some who think that the operator's is hard to read, but with practice it is as natural as all other operators.

Methods return values

The methods described in the previous example, all have a return type which is either *int* or *double*. When a method has a return type, it has a value after it has been executed. The method can therefore be part of an expression in the same way as a variable. For example one can write the following:

```
double x = Max(3.14, 1.41);
```

which means that the return value of method *Max()* is stored in the variable *x*. Once a method has a return type, it ends with a return statement, which determines the return value – often as a result of an expression. There is not much to say concerning methods return types, except that the return type can be anything – including reference types – and that a method can have only one return value. As an example is shown a simple input method that returns a string:

```
static string Indtast(string text)
{
   Console.Write(text + ": ");
   return Console.ReadLine();
}
```

As a last remark concerning return types it must also include *void* that is not a type, but simply a term for a method that has no return value. Although the method is *void*, it may well have an empty *return* statement, which did not gives the method a value, but only terminate the method.

Download free eBooks at bookboon.com

Properties

A property is, as mentioned earlier really just a method with a different syntax and, in principle, a property can do anything. The idea of a property is, however, that it read (returns) the value of an instance variable and possible changes its value. In addition, you should use the convention that the name of a property is the variable's name written in uppercase.

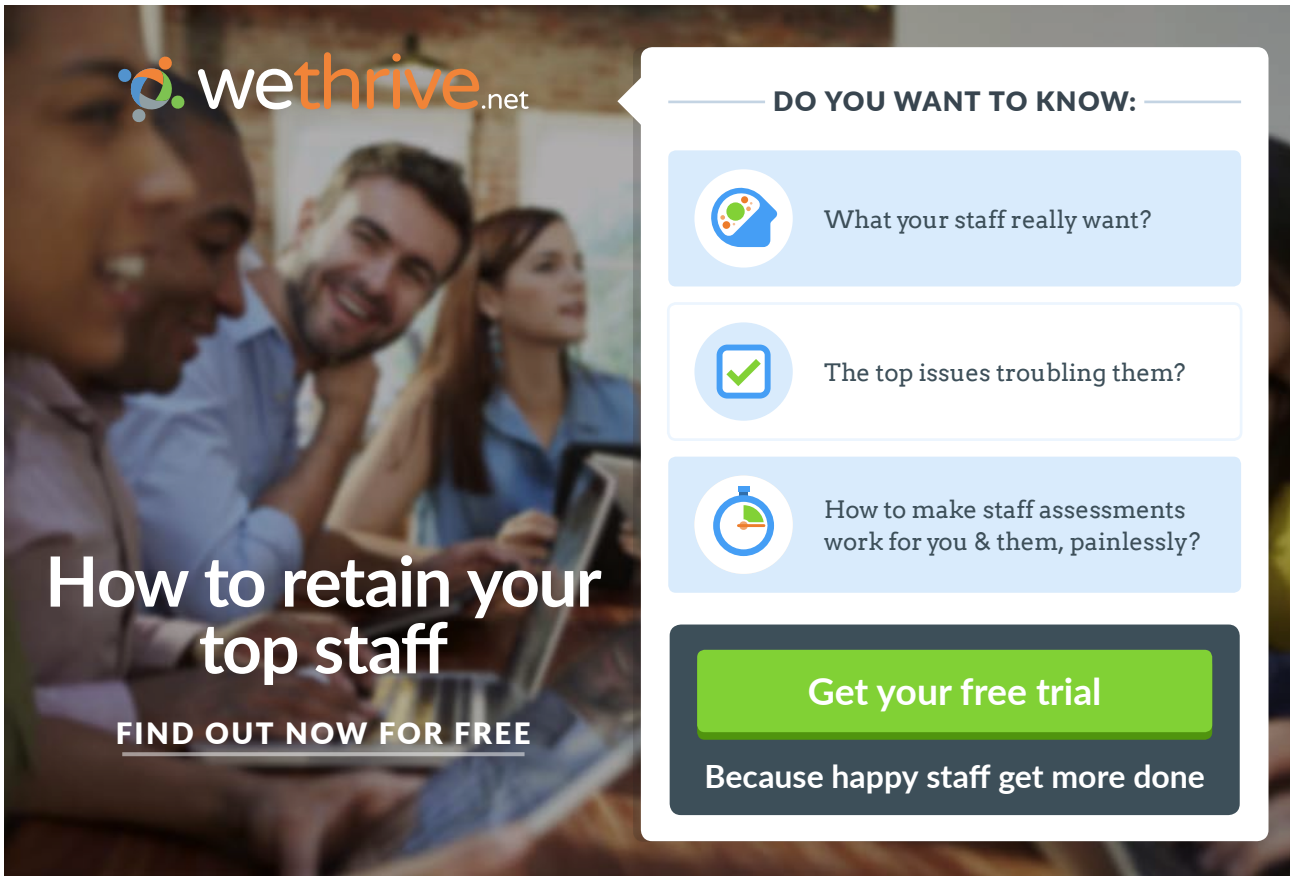
Exam25

A point

This example illustrates a simple class representing a point in a coordinate system.

How to

```
class Point
{
   private int x;
   private int y;

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



Click on the ad to read more

Download free eBooks at bookboon.com

```
   public int X
   {
      get { return x; }
      set { x = value; }
   }

   public int Y
   {
      get { return y; }
      set { y = value; }
   }

   public override string ToString()
   {
      return string.Format("({0},{1})", x, y);
   }
}
```

Explanation

The coordinates are *private*, and for that you can't access them from outside the class. To do so you must define *get* properties. If it also should be possible to modify an object's coordinates, the class must also define *set* properties, for example

```
set { x = value; }
```

The meaning is that you can change the value of the variable *x*, but the syntax is a bit special, corresponding to the value to be assigned to *x* is represented by the reserved word *value*. The result is that, in a program you can be write as follows:

```
Point p = new Point(2, 3);
p.X = 5;
```

where the x-coordinate gets a different value.

When a class as above offers both *get* and *set* properties, and *set* properties do nothing more than to assign the value of *value*, so there is no difference to simply define both variables as public. It is advisable, nevertheless, to comply with the definition of variables private and then the necessary properties, since, as mentioned that means, that it is up to the programmer, for unlocking the protection. There will often be assigned limitations to *set* properties in a class. If, for example you assume that the x-coordinate in the above example must always be between 0 and 1023, you could possibly write the *set* method as follows:

```
set { if (value >= 0 && value <= 1023) x = value; }
```

As a final comment on the above class, you should note the constructor. The class has two instance variables named, respectively *x* and *y*, and the constructor has two parameters which are also called the *x* and *y*. It gives inside the constructor a name coincidence problem, since there are two things (instance variable and parameter) with the same name. The problem is solved with the word *this*, where *this.x* means the instance variable, while *x* (with nothing) means the parameter.

Download free eBooks at bookboon.com

Parameters

A method's parameters can be of any type, but there are several kinds of parameter passing. When you define a method that you specify the parameters the method should have, and these parameters are called the *formal parameters* and specify the values that the method should operate on. When you call the method, you must specify the values (parameters) to be transferred to the method, and they are called for the *actual parameters*.

Value parameters

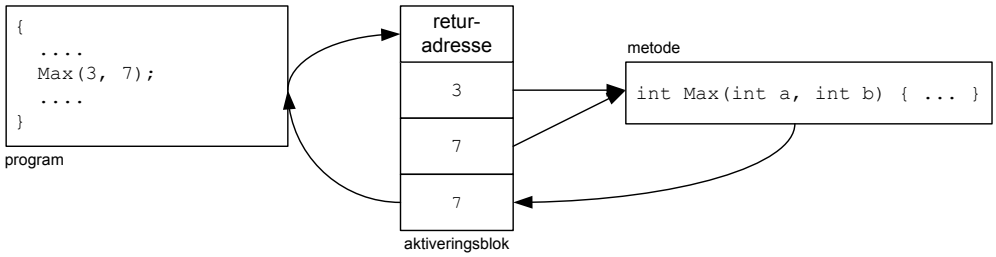
In general the parameters are value parameters, for example

```
static int Max(int a, int b)
{
   return a < b ? b : a;
}
```

When this method is called, there must be transmitted two actual parameters. Exactly what happens is that on the stack the system creates a so-called activation block, which contains four basic things:

- the return address, so the system knows where the program will continue after the method is terminated
- a copy of each actual parameter
- a place to the return value corresponding to the method's type
- the method's local variables, if it has local variables

One can outline it as follows:



When the program calls the method *Max()* it creates an activation block on the stack, and the return address and the actual parameters are copied to it. Next the method takes the control and starts its work. The method then always works on the copies (*a* and *b*) located on the stack, and it also means that if the method changes these values it alters the copies on the stack, and these changes has no effect in the program. When the method performs its return statement, the return value is copied to the activation block, then the method terminates and the control is given back to the program and it continues its work. Immediately after the program has gained back the control, it can read the return value from the activation block, which will then be removed from the stack. Note that this also means that if the method creates local variables (which are not the case in this example) they are also gone.

Download free eBooks at bookboon.com

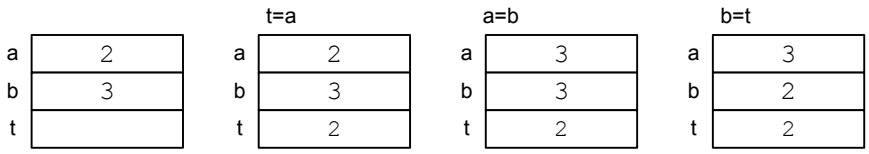
Consider as another example, the following method:

```
public void Swap(int a, int b)
{
   int t = a;
   a = b;
   b = t;
}
```

When done, it swap the values of the two parameters *a* and *b*. If you test the method *Swap()*, you will see that the current parameters are not reversed:

```
static void Test2()
{
   int t1 = 2;
   int t2 = 3;
   Swap(t1, t2);
   Console.WriteLine("{0} {1}", t1, t2);
}
```

If you look at what *Swap()* creates on the stack, it will swap the *a* and *b*



Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

but when it happens in the activation block on the stack, the two variables *t1* and *t2* are unchanged after the *Swap()* is executed.

It should however be aware of what it means to reference types, since the result here may be different than expected. Consider the example of the class *Point*, and the following method, which has a *Point* as a parameter:

```
static void Mul(Point p, int t)
{
   p.X *= t;
   p.Y *= t;
}
```

If you execute the following statements:

```
Point p = new Point(2, 3);
Mul(p, 2);
Console.WriteLine(p);
```

you get the result

(4,6)

and thus the method *Mul()* has changed the *Point* object. Just it does not seem as the parameter is transferred as copy on the stack. The method *Mul()* has two parameters, and here is the type of the first parameter *Point* and thus a class. The actual parameter is thus a reference, and is placed on the stack, which means that the *Mul()* method refers to the same object as the program. As a result, it is this object that is changed.

If you change the *Mul()* method to the following

```
public void Mul(Point p, int t)
{
   p = new Point(p.X, p.Y);
   p.X *= t;
   p.Y *= t;
}
```

the method will create a new *Point* object and the reference on the stack will refer to this object. When *Mul()* terminates, its activation block is removed, and the reference to the new object is gone, and thus the change that *Mul()* performs. The result is that the object to which the program refers to is unchanged, and the program will write

(2,3)

So there is reason to pay attention to what happens if you transfer references as parameters to a general value parameter.

Download free eBooks at bookboon.com

Reference parameters

In .NET you can also use true *reference parameters*. For example you can change the method *Swap()* for the following:

```
public void Swap(ref int a, ref int b)
   {
   int t = a;
   a = b;
   b = t;
}
```

If you then change the method *Test2()* so that the actual parameters are references

```
static void Test3()
{
   int t1 = 2;
   int t2 = 3;
   Swap(ref t1, ref t2);
   Console.WriteLine("{0} {1}", t1, t2);
}
```

it are references to the actual parameters that are placed on the stack, and *Swap()* will swap the values of the variables *t1* and *t2*. That means that the changes *Swap()* performs, is on the objects that the references on the stack applies to.

Reference parameters are used instead of value parameters when the changes that a method do on its parameters must be maintained in the calling code.

Note that you can combine value parameters and reference parameters so that the method may have parameters for both references and values.

out-parameters

In .NET, there is a possibility of using *out parameters*, which are parameters which may not be initialized before a method is called, and instead are assigned a value in the method. The following method creates two *Point* objects assigned to references that are passed as *out* parameters to the method:

```
static void Points2(int x1, int y1, int x2, int y2, out Point p1, out Point p2)
{
   p1 = new Point(x1, y1);
   p2 = new Point(x2, y2);
}
```


Download free eBooks at bookboon.com

If the method is performed as follows:

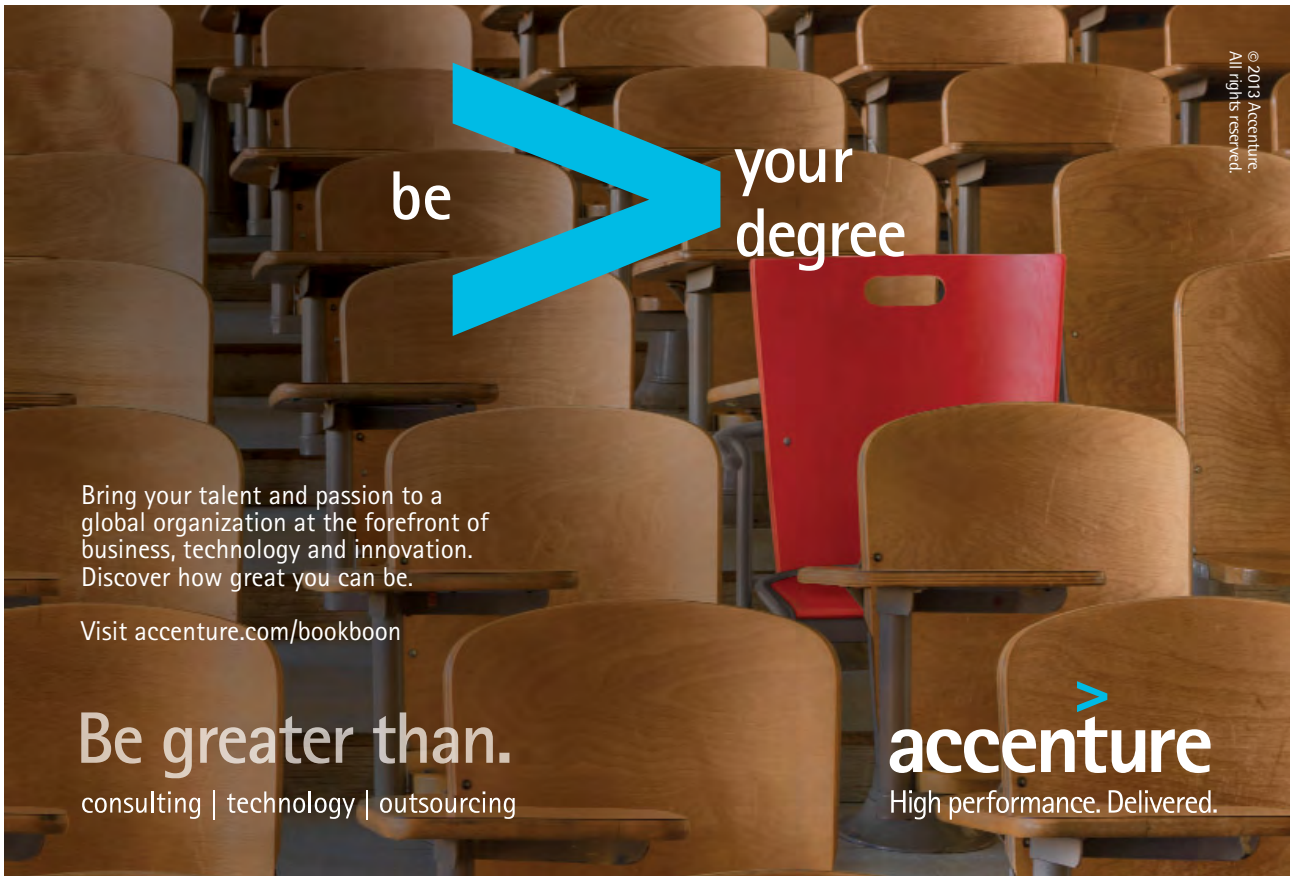
```
static void Test4()
{
   Point p1;
   Point p2;
   Points2(2, 5, 7, 3, out p1, out p2);
   Console.WriteLine(p1);
   Console.WriteLine(p2);
}
```

then *p1* and *p2* refer to the objects that are created in the method *Points2()*. Note that *p1* and *p2* is not initialized in the program, which is not necessary, since they are used as actual *out* parameters. A method can have only one return value and *out* parameters solves a problem, where it is desirable that a method must return multiple values. For example the method *Points2()* returns two values, which is not possible, and the problem can then be solved by giving the method two *out* parameters.

Default parameters

It is also possible to set default values for the parameters. The following method has three parameters:

```
static double Calculate(double price, int units = 1, double discount = 5)
{
   return price * units * (100 - discount) / 100;
}
```



Click on the ad to read more

Download free eBooks at bookboon.com

The method calculates the price of goods excl. discount. The number of units is by default set to 1, and discount rates are as default 5 percent. This means that the method can be called without specifying values for these parameters:

```
static void Test5()
{
   Console.WriteLine(Calculate(20));
   Console.WriteLine(Calculate(20, 5));
   Console.WriteLine(Calculate(20, 10));
}
```

and if you don't, the default values are used. In principle, all parameters can have a default value, but the parameters that have a default value must be last.

Variable number of parameters

As a last remark concerning parameters, I will show how it is possible to have a method with a variable number of parameters. The following method has an array of the type *string* as a parameter, and the method returns a string consisting of all strings in the array separated by spaces:

```
static string Concat(params string[] text)
{
   if (text.Length == 0) return "";
   string temp = text[0];
   for (int i = 1; i < text.Length; ++i) temp += " " + text[i];
   return temp;
}
```

What is important is the word *params*, which means that one can set a variable number of actual parameters. For example the method can be called as follows:

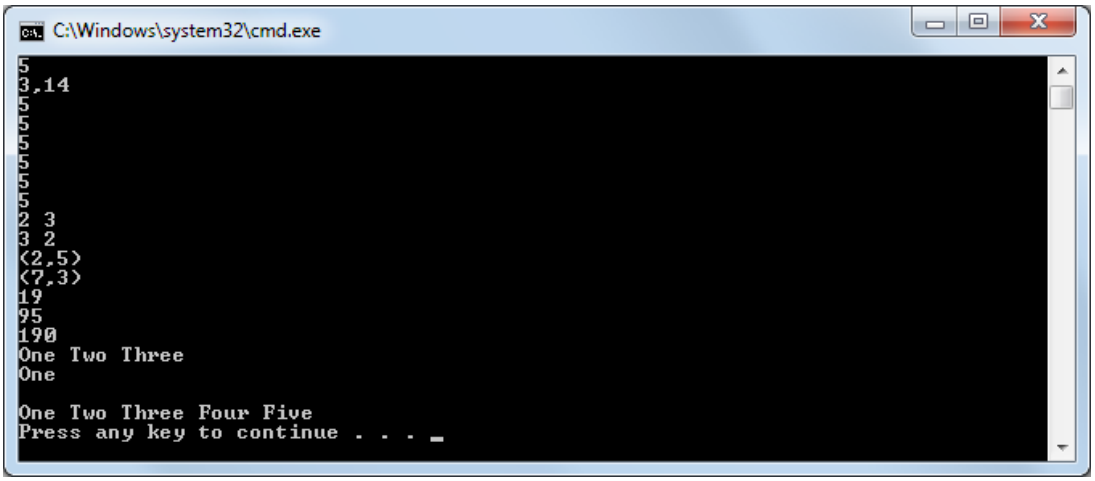
```
static void Test6()
{
   Console.WriteLine(Concat("One", "Two", "Three"));
   Console.WriteLine(Concat("One"));
   Console.WriteLine(Concat());
   Console.WriteLine(Concat("One", "Two", "Three", "Four", "Five"));
}
```


Download free eBooks at bookboon.com

Exam26

Methods parameters

All of the above test methods about methods parameters are combined in the example Exam26 and if you run this example, you get the following results:



Unlock your potential
eLibrary solutions from bookboon is the key
bookboon eLibrary
Interested in how we can help you?
email ban@bookboon.com
Click on the ad to read more

Download free eBooks at bookboon.com

11 Inheritance

A class must, as mentioned several times represent a thing in the program's problem domain, and in the design you decide which properties a class must have. When a class is first completed, tested and put into operation one must be careful to open it again and make changes when there is a significant risk that changes may have unintended consequences for programs that use the class. But no matter how careful you are under design, there is a high probability that it is necessary to extend a class with new properties, and this is where inheritance comes into play as a technique to extend a class without changing the existing class. Another thing that inheritance must address is the situation where you have two classes that are similar but also have differences. You can sometimes put what they have shared in a base class that the other must inherit

Exam31

Points

This example is merely to show the syntax, but is not an example of where inheritance is used in practice.

The following class defines a point in a coordinate system:

```
public class Point
{
   private int x;
   private int y;

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

   public int X
   {
      get { return x; }
      set { x = value; }
   }

   public int Y
      {
      get { return y; }
      set { y = value; }
      }

   public override string ToString()
   {
      return string.Format("({0}, {1})", x, y);
   }
}
```


Download free eBooks at bookboon.com

The class is simple and requires no further explanation. I will now write a class that inherits *Point*, and one can think of it as a class that extends the class *Point* with new methods or properties – in this case only a single method:

```
public class NewPoint : Point
{
   public NewPoint(int x, int y) : base(x, y)
   {
   }

   public void Add(Point p)
   {
      X += p.X;
      Y += p.Y;
   }
}
```

The syntax for inheritance is:

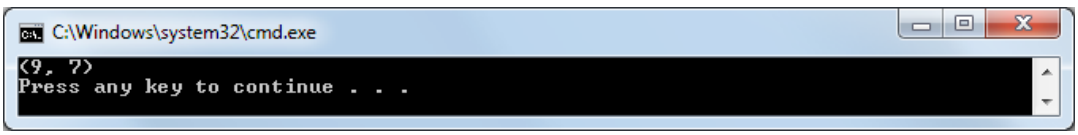
```
public class NewPoint : Point
```

which means that *NewPoint* inherits or extends *Point*. The result is that a *NewPoint* object has the *pubilc* properties (properties and methods) as *NewPoint* defines, together with the *public* characteristics the *Point* class defines. The idea is that if a class needs some new features, then you inherit rather than modify the existing class. That way you avoid changing the classes that are already created and in use and that you know works.

When you create a *NewPoint* object, it must transfer values to the class that it inherited from, what is done with *base* after the constructor. In fact, it corresponds to that the constructor of the class *Point* is performed.

Below is a program which uses the *NewPoint* class:

```
static void Main(string[] args)
{
   NewPoint p1 = new NewPoint(2, 3);
   NewPoint p2 = new NewPoint(1, 4);
   p1.Add(p2);                          // metode defineret i NewPoint
   p1.X = p1.X * 3;                     // property defineret i Point
   Console.WriteLine(p1);               // ToString() metoden fra Point
}
```



Download free eBooks at bookboon.com

Exam27

Persons

As an example, I will start with a class that represents a person by a first and last name:

```
public class Person
{
   private string firstName;
   private string lastName;

   public Person(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;
   }
}
```

The class is very simple and does not require many comments.

I will then define a class that represents an employee (in a company) when an employee is a person with two additional properties in the form of a title and a salary. The class *Employee* can be written in several ways, and you can for example think of it as an extension of the class *Person* with two new properties. The solution is not just to extend the *Person* class, as you can imagine this class is used in contexts in which concepts such as title and salary are not meaningful. Instead, you can write a class that inherits the *Person* class:

```
public class Employee : Person
{
   private string position;
   private int monthly;

   public Employee(string firstName, string lastName, string position, int monthly)
      : base(firstName, lastName)
   {
      this.position = position;
      this.monthly = monthly;
   }

   public string Position
   {
```


Download free eBooks at bookboon.com

```
      get { return position; }
   }

   public int Monthly
   {
      get { return monthly; }
   }

   public override string ToString()
   {
      return base.ToString() + "\n" + position;
   }
}
```

First, notice after the name of the class *Employee* I have written

```
: Person
```

It is telling that *Employee* inherits *Person*. That *Employee* inherits *Person* means an *Employee* gets (inherits) all the properties that a *Person* has and expands with new properties. In this case extends an *Employee* a *Person* with two new instance variables and associated properties. The *Employee* class has a constructor that initializes the two new instance variables, *position* and *monthly*, but it must also initialize the instance variables *firstname* and *lastname* in the base class. This requires that the constructor in *Person* is called, but when as a programmer you can't call it directly, it is necessary to have a syntax that performs the constructor in the base class. It happens in the following:

```
: base(firstName, lastName)
```


Click on the ad to read more

Download free eBooks at bookboon.com

Here, the colon and *base()* after constructor declaration means that the constructor in the *Person* is performed by *firstName* and *lastName* as the actual parameters. Specifically, what happens is when you create an *Employee* object the constructor in *Person* is performed, and then the constructor in *Employee* is performed. When an *Employee* is specifically a *Person*, one can say that *Person* has to be created first, before the *Employee* can be created. An *Employee* object can then refer to the two properties *FistName* and *LastName*. An *Employee* object can refer to *FirstName*, since it is a public property in *Person* and because *Employee* inherits *Person*. *Employee* objects can refer to all public members in both *Person* and *Employee*. Private members can still only be referenced from within the class where they are defined. For example the variable *firstname* in *Person* can't be referred from methods in *Employee*.

I will then define a class that represents a director. A *Director* is just a special kind of *Employee*, and the class can be defined as a class that inherits *Employee*:

```
public class Director : Employee
{
   public Director(string firstName, string lastName, int monthly)
      : base(firstName, lastName, "Director", monthly)
   {
   }
}
```

The class is very simple and consists solely of a constructor, which transmit parameters to the constructor in the *Employee* class. All the other services as a director does occur (inherited) come from *Employee* and *Person*.

At exactly the same way one can define a class that represents a bookkeeper:

```
public class Bookkeeper : Employee
{
   public Bookkeeper(string firstName, string lastName, int monthly)
      : base(firstName, lastName, "Bookkeeper", monthly)
   {
   }
}
```

Now consider the following method:

```
static void Print(Employee e)
{
   Console.WriteLine(e);
   Console.WriteLine("Monthly: {0}", e.Monthly);
   Console.WriteLine("{0}, {1}", e.LastName, e.FirstName);
}
```

The first statement prints the result of *ToString()* from the class *Employee*. The next one is not much mystery in, but you should note the last, which uses two properties, both of which are defined in *Person*. The parameter e has the type *Employee*, and you can thus especially use what is defined *public* in *Person*.

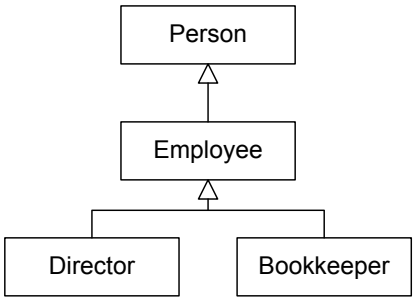
Download free eBooks at bookboon.com

Consider also the following method, which creates two objects, respectively of the types *Director* and *Bookkeeper*:

```
static void Test1()
{
   Director d = new Director("Olga", "Jensen", 8000);
   Bookkeeper b = new Bookkeeper("Karlo", "Hansen", 5000);
   Print(d);
   Print(b);
}
```

You should particularly note that the method calls *Print()* with the two objects *d* and *b* as actual parameters. It makes sense for *d* have the type *Director*, which specifically is an *Employee*.

Above, there are defined four classes that are linked in a hierarchy, as you can illustrate in the following way:



When a class inherits another class, the class you inherit from is called the *base class* and the inheriting class is called a *derived class*. For example is *Person* the base class for *Employee* while *Employee* derives from *Person*. Sometimes *Person* instead of is called a *super class*, while the *Employee* is called a *subclass*. We say also that *Person* is a *generalization* of *Employee* and that *Employee* is a *specialization* of *Person*. This saying better reflected the relationship between *Employee*, *Director* and *Bookkeeper* in which *Employee* is a generalization of the *Director* and *Bookkeeper* and *Director* are specializations of *Employee*. Sometimes we talk also about the class that inherits as an extension of the base class corresponding to *Employee* extends *Person* with new properties.

I will write another class, representing a consultant that is an *Employee* whose salary is calculated as a fixed monthly salary and a commission on the sale:

```
public class Consultant : Employee
{
   private double commission;
   private double sale;

   public Consultant(string firstName, string lastName, int monthly,
      double commission) : base(firstName, lastName, "Consultant", monthly)
   {
      this.commission = commission;
   }
```


Download free eBooks at bookboon.com

```
   public double Commission
   {
      get { return commission; }
   }

   public double Sale
   {
      get { return sale; }
      set { sale = value; }
   }

   public override int Monthly
   {
      get { return monthly + (int)(sale * commission / 100); }
   }
}
```

Note first that a *Consultant* extends the class *Employee* with two new variables. One is initialized in the constructor, whereas the other gets a value by means of a property. There is not much to say. But if you look at the property *Monthly*, you should note two things: the word override, and that it refers to the variable *monthly* in the base class *Employee* – which is not possible because it is *private*. The problem is solved by changing the class *Employee* and instead defines the variable *monthly* as *protected*:

```
public class Employee : Person
{
   protected int monthly;
```

Click on the ad to read more

Download free eBooks at bookboon.com

If a member of a class is *protected*, it can be referenced by derived classes, as it was *public* while it from objects of this class looks as *private*. *protected* is a visibility between *private* and *public*, which allows derived classes to refer to members in the base class, while the base class maintains protection against other classes. Stated differently, then variable *monthly* may be referenced by subclasses of *Employee*, but not from classes that do not inherit *Employee*. A class should not make all of its variables *protected*, but only those variables that one must expect that the derived classes need to refer to. When you make a member *protected* you also are opening up the protection in relation to derived classes.

In this case, *protected* is only included to explain the concept, the problem could be solved in another way:

```
public override int Monthly
{
   get { return base.Monthly + (int)(sale * commission / 100); }
}
```

where you with *base* refers to the property *Monthly*, which is defined in the base class.

There is now a definition of the property *Monthly* in both *Employee* and *Consultant*, and the meaning is that in *Consultant* shall override the property in *Employee* – give it a different meaning. For that to be possible, you must in *Employee* open up for it by declaring the property *virtual*:

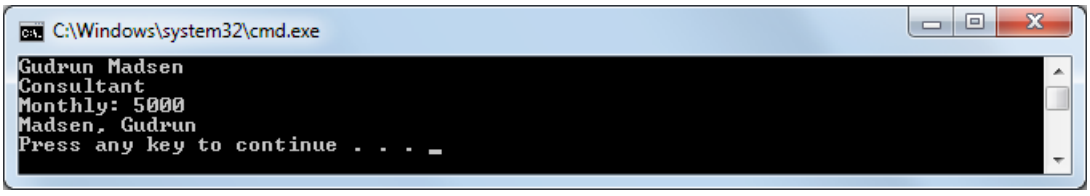
```
public virtual int Monthly
{
   get { return monthly; }
}
```

When that is the case, a derived class – here *Consultant* – can choose to override the property by entering the keyword *override*. If you do not, you get a warning that it hides the base class version.

It was the class *Consultant*, and below is shown a method that uses the class:

```
static void Test2()
{
   Consultant c = new Consultant("Gudrun", "Madsen", 2000, 10);
   c.Sale = 30000;
   Print(c);
}
```

If you run the method, you get the result:



Download free eBooks at bookboon.com

It is not entirely obvious. Note that the property that is executed for *Monthly*, is the one in the class *Consultant*, even though the parameter to the *Print()* method is of the type *Employee*. This means that the system "remembers" the type of the current object, even if the object in *Print()* is known as an *Employee*. It is an extremely important option known as *polymorphism*.

When you write a class, you have no guarantee that there not in the future is one that inherits the class and extends it with new features – and that is exactly also the idea of inheritance. However, there may be situations where you do not want this option and you can then declare the class *sealed*, meaning that it can't be inherited. If for example you do not want it to be possible to inherit the class *Director*, you can define it as follows:

```
public sealed class Director : Employee
{
   public Director(string firstName, string lastName, int monthly)
      : base(firstName, lastName, "Director", monthly)
   {
   }
}
```

Comment

Inheritance is not an especially difficult concept – at least not when you've seen it a few times – but there are certain things one must be aware of:

- A class – for example *Employee* – may have one or more derived classes, but a class can inherit only one class.
- Polymorphism – that the runtime system remembers the specific type of an object (the type which is used when the new object is created – is one of the most important concepts of object oriented programming.

Moreover, there are some names associated with inheritance, that it is important to understand:

- base
- protected
- virtual
- override

You can in the above examples to see how these words are used.

Download free eBooks at bookboon.com

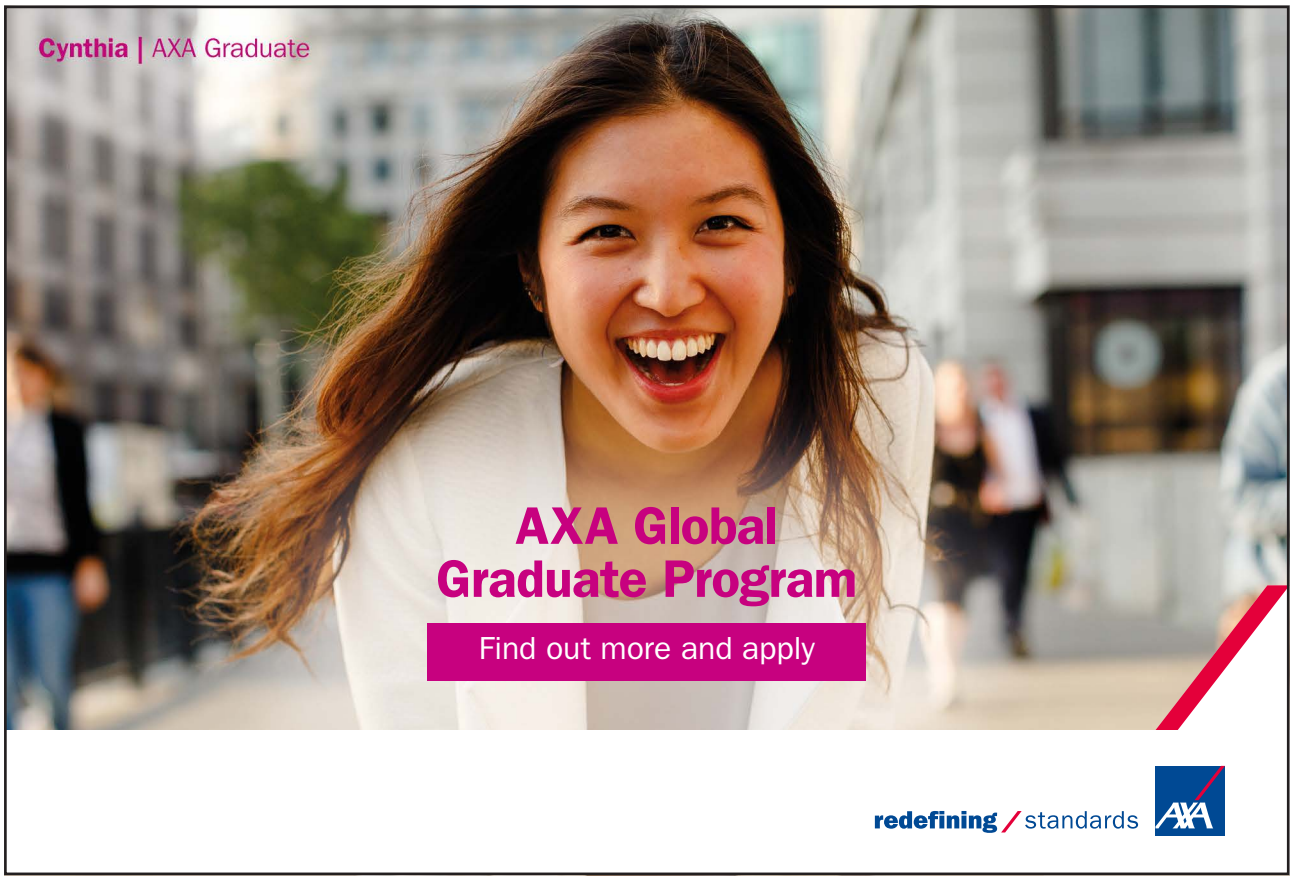
12 The class Object

I have above seen on some classes such as *Coin*, *Dice*, *Cup* etc. and these classes have not used inheritance – apparently, but actually, they have indirectly inherits a class called *Object*. If you do not write anything, then any class automatically inherit this class, and thus all classes without exception, directly or indirectly inherit *Object*. It is actually more than that, for any type whether it is a value type or reference type inherits *Object*.

C# defines an alias *object* of class *System.Object*, which is a reserved word, exactly the same way as the *string* is an alias for the class *System.String*.

The class *Object* does not contain much, and its primary purpose is to be a common base class for all types. The class defines a few methods that I will mention below. First I will mention *ToString()* which returns the value of an object as a string. Consider the following class:

```
class ZipCode
{
   private string code;
   private string name;
```



Click on the ad to read more

Download free eBooks at bookboon.com

```
   public ZipCode(string code, string name)
   {
      this.code = code;
      this.name = name;
   }

   public string Code
   {
      get { return code; }
   }

   public string Name
   {
      get { return name; }
   }
}
```

When not specified otherwise, the class *ZipCode* inherits the class *Object*. Consider the following method:

```
static void Test1()
{
   ZipCode z = new ZipCode("7800", "Skive");
   Console.WriteLine(z);
}
```

which created an object of type *ZipCode*, and this object is printed on the screen. Note that when the object is printed with *WriteLine ()*, then it is the result of the object's *ToString()* method that is printed. In this case there is no *ToString()* method in class *ZipCode*, but the program can be translated and run, and the result is the following:

The *ToString()* method that is carried out, comes from the class *Object*, and print the full name of the object's type, here is the class's namespace and class name. One can thus state that all objects without exception has a *ToString()* method, but it is the one who writes the object's class, which is responsible for override the method so that it returns a meaningful result. As a class always inherits *Object*, that is why I've written *override* in all the classes that have defined a *ToString()* method – *ToString()* is a *virtual* method in the class *Object*.

Object also defines a method called *Equals()* that have an object as a parameter. It is a method that returns *true* if the current object and the parameter are the same. If you execute the following method

```
static void Test2()
{
   ZipCode z1 = new ZipCode("7800", "Skive");
   ZipCode z2 = new ZipCode("7800", "Skive");
   Console.WriteLine(z1.Equals(z2));
}
```


Download free eBooks at bookboon.com

it will write *False* on the screen, and it was not what one would expect. The two objects *z1* and *z2* have the same value, but they are two different objects, each of which refers to their own object on the heap. The method *Equals()* as defined in the class *Object*, compares the object references, and as *z1* and *z2* are two different objects will *Equals()* returns false, even if the two objects have the same value. Should it be otherwise, it is up to the programmer to override the method *Equals()*, so it compares the values rather than references. If you want to override *Equals()* in the class *ZipCode* you can do the following:

```
public override bool Equals(object obj)
{
   if (obj is ZipCode)
   {
      ZipCode z = (ZipCode)obj;
      return code.Equals(z.code) && name.Equals(z.name);
   }
   return false;
}
```

Note first the *is* operator that may be used to test whether an object has a particular type. For *Equals()* should return *true*, *obj* must be at least of the type *ZipCode*. If so, you can type cast it to a *ZipCode*. Then *obj* is equal to the current object if both the zip code and the city name are the same. Note that this test is in fact based on that the *string* class overrides *Equals()* with value semantic.

I will mention another method in the class *Object* called *GetHashCode()*. It is a method of an object that returns an integer that can be perceived as an identification of the object. In general, this code is determined from the reference to the object, but it is in the same manner as for *ToString()*, and *Equals()* up to the programmer to override the method, if it has to return a code determined from the value of the object. There are different guidelines for how this code is to be determined, but you can observe that it is not a requirement that two different objects return different hash codes. However, it should be the case that if you have two objects *obj1* and *obj2* and *obj1.Equals(obj2)* is true, then must *obj1.GetHashCode()* be equal to *obj2.GetHashCode()*. I will not at this point to give examples of applying this method, but the examples will come later. Another reason to mention *GetHashCode()* on this point is that if you overrides *Equals()* without having to override *GetHashCode()*, you get a warning from the compiler. In most cases, this warning could be ignored.

Consider the following method, where you should be especially noted that the method has an object as a parameter:

```
static void Print(object obj)
{
   Console.WriteLine(obj.GetType());
   Console.WriteLine(obj.GetHashCode());
   Console.WriteLine(obj);
}
```


Download free eBooks at bookboon.com

You should also note the method *GetType()*, which is also a method in the class *Object*, which in this case is used to print the name of the object's type. The following method creates an array of the type *object* and prints its elements on the screen:

```
static void Test3()
{
   object[] t = { 23, 3.14, "Volmer", new Postnummer("7800", "Skive") };
   for (int i = 0; i < t.Length; ++i) Print(t[i]);
}
```

If you run the method you get the following result:

You should primarily note two things:

- a method that as a formal parameter has an *object* may have an actual parameter of any type
- an array which type is *object* may contain anything irrespective of the type

At first glance it sounds smart, but you should be aware that this means that the compiler can't type check, and thus a code based on the type *object* can very easily contain errors.

The above example concerning the class *Object* is called Exam28.

Download free eBooks at bookboon.com

13 Abstract classes

In this chapter I will introduce the concept of an abstract class. It is a class that can't be instantiated – you can't create objects of an abstract class, but abstract classes can for example be base classes for other concrete classes.

Exam32

Abstract points

I will again start with an example that has the sole purpose of showing the syntax and the example is essentially the same as Exam31. The class *Point* is now *abstract*:

```
public abstract class Point
{
   private int x;
   private int y;

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

Click on the ad to read more

Download free eBooks at bookboon.com

```
   public int X
   {
      get { return x; }
      set { x = value; }
   }

   public int Y
   {
      get { return y; }
      set { y = value; }
   }

   public override string ToString()
   {
      return string.Format("({0}, {1})", x, y);
   }

   public abstract void Dec();
   public abstract void Inc();
}
```

There are two differences. The class now has two abstract methods *Dec()* and *Inc()*. These methods have no code and thus are merely definitions. This means that those who use the class *Point* knows that these methods exist, but not how they work. The second difference is that the class itself is defined *abstract*. It must be, and that means that the class can't be instantiated – you can't create objects of the type *Point*

The class *NewPoint* inherits *Point*, and must therefore implement the two abstract methods, so they are specific methods that perform something:

```
public class NewPoint : Point
{
   public NewPoint(int x, int y) : base(x, y)
   {
   }

   public override void Dec()
   {
      --X;
      --Y;
   }

   public override void Inc()
   {
      ++X;
      ++Y;
   }
}
```

Note that an abstract method that is implemented in a derived class has to be preceded with the word *override*.

Download free eBooks at bookboon.com

Below is a *Main()* method, which uses the class *NewPoint*:

```
static void Main(string[] args)
{
   Point p = new NewPoint(2, 3);
   Console.WriteLine(p);
   p.Inc();
   Console.WriteLine(p);
   p.Dec();
   Console.WriteLine(p);
}
```

p is a *NewPoint* object, but it is in the program defined as a *Point*. It is possible as a *NewPoint* specifically is a *Point*, but you should particularly note that even if *p* is a *Point*, you can still write

```
p.Inc();
```

Since *Point* has an abstract method *Inc()*, the compiler knows that a *Point* object has such a method (although it is defined in a derived class), so the above statement makes sense.

Loan

As an example I will write a class that can represent a loan in a bank. A loan is (in this example) characterized by a principal, a number of periods and an interest rate. I would assume that the interest rate remains constant throughout the payment period, and there is one payment every period on due date and that the payment falls one period after the loan inception. I would also assume that you are interested in the following information

- the nth payment
- repayment at the nth payment
- interest at the nth payment
- outstanding after the nth payment have been paid

The finished program must print a plan for the loan's amortization that is an overview of how the loan looks after each period.

Download free eBooks at bookboon.com

How to

Corresponding to this a loan could be represented by the following class:

Loan
principal: double rate: double periods: int
Principal: double Rate: double Periods: int Payment(n): double Interest(n): double Repayment(n): double Outstanding(n):double

Now there are several kinds of loans, and as an example one can look at a serial loan that is characterized by the fact that you for each payment pays the same in repayment plus interest on it at any time due. At such a loan the payment decreases over the repayment period. If, for example you have a serial loan of $1000 and there are 10 periods, you have to pay 10 payments where the repayment each time is $100.

Click on the ad to read more

Download free eBooks at bookboon.com

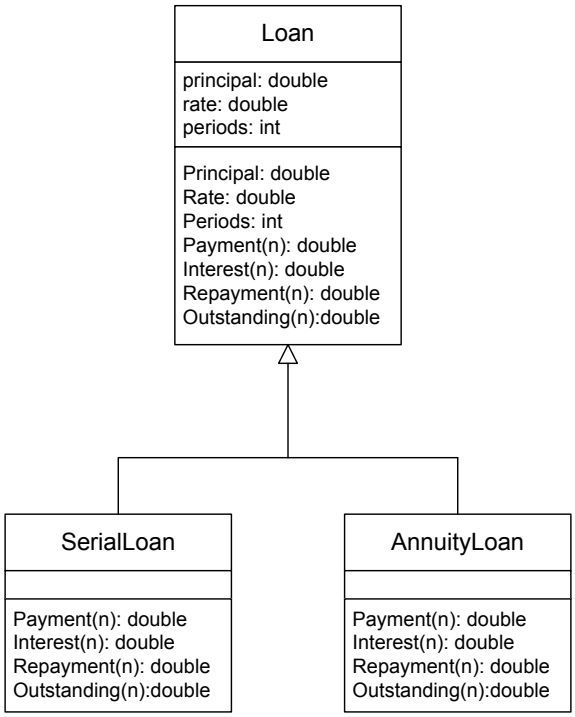
You can also look at an annuity, which is characterized in that the payment is constant throughout the repayment period. This means that the relationship between interest and repayment are so that in the beginning the interest is a big part of the payment and the repayment is only a small part, and towards the end of the period, the situation is reversed. The relationship between the loans principal, payment and the interest rate can be expressed in the following formula:

$$principal = payment\frac{1-(1+rate)^{-periods}}{1+rate}$$

Finally, the outstanding immediately after the payment of the nth payment is calculated as:

$$restloan = principal(1+rate)^{n} - payment\frac{(1+rate)^{n}-1}{rate}$$

If you look at the above class *Loan* there is a need for two versions: One for a serial loan and one for an annuity. When two classes have much in common, it is natural to think of a design with a common base class and two derived classes:



The three classes can be implemented in the following manner:

```
public abstract class Loan
{
   protected double principal;
   protected double rate;
   protected int periods;
```


Download free eBooks at bookboon.com

```
   public Loan(double principal, double rate, int periods)
   {
      this.principal = principal;
      this.rate = rate;
      this.periods = periods;
   }

   public double Principal
   {
      get { return principal; }
   }

   public double Rate
   {
      get { return rate; }
   }

   public int Periods
   {
      get { return periods; }
   }

   abstract public double Payment(int n);
   abstract public double Interest(int n);
   abstract public double Repayment(int n);
   abstract public double Outstanding(int n);
}

public class SerialLoan : Loan
{
   public SerialLoan(double principal, double rate, int periods) :
      base(principal, rate, periods)
   {
   }

   public override double Repayment(int n)
   {
      return principal / periods;
   }

   public override double Outstanding(int n)
   {
      return Repayment(0) * (periods - n);
   }

   public override double Interest(int n)
   {
      return Outstanding(n - 1) * rate;
   }

   public override double Payment(int n)
   {
      return Repayment(n) + Interest(n);
   }
}

public class AnnuityLoan : Loan
{
   public AnnuityLoan(double principal, double rate, int periods) :
      base(principal, rate, periods)
   {
   }
```


Download free eBooks at bookboon.com

```
   public override double Payment(int n)
   {
      return principal * rate / (1 - Math.Pow(1 + rate, -periods));
   }

   public override double Outstanding(int n)
   {
      return principal * Math.Pow(1 + rate, n) - Payment(0) *
         (Math.Pow(1 + rate, n) - 1) / rate;
   }

   public override double Interest(int n)
   {
      return Outstanding(n - 1) * rate;
   }

   public override double Repayment(int n)
   {
      return Payment(n) - Interest(n);
   }
}
```

The program must print a plan for the loan's amortization, which is a table showing the status of the loan each payment:

```
public class Amortisation
{
   private Loan loan;
```

Click on the ad to read more
Download free eBooks at bookboon.com

```
   public Amortisation(Loan loan)
   {
      this.loan = loan;
   }

   public void Print()
   {
      Console.WriteLine("Principal: {0, 10:F}", loan.Principal);
      Console.WriteLine("Rate of interest: {0, 10:F}", loan.Rate);
      Console.WriteLine("Number of periods: {0, 10:D}\n", loan.Periods);
      Console.WriteLine(
         "Periods Payment Repayment Interest Outstanding");
      for (int n = 1; n <= loan.Periods; ++n)
         Console.WriteLine("{0, 7:D}{1, 15:F}{2, 15:F}{3, 15:F}{4, 15:F}", n,
            loan.Payment(n), loan.Repayment(n), loan.Interest(n), loan.Outstanding(n));
   }
}
```

Explanation

Note first that the class *Loan* is defined *abstract*. It must be because it has abstract methods. An abstract method is a method that is not defined – its code is not written. In this case there are four abstract methods. The reason that the methods are abstract is that how for example the payment is calculated depends on what kind of loan it is. The method can only be implemented when you know what kind of loan you are talking about. An abstract method is simply a prototype definition in which a method is defined by its name, return type and parameters, but its body is missing.

Loan is a type, and even though it is abstract, and you thus can't create objects of type *Loan*, the type can for example be used as a type of an argument to a method. Everybody who gets the informed that they are dealing with something of the type *Loan* knows that it is something that beyond the three properties have four methods, *Payment()*, *Interest()*, *Repayment()* and *Outstanding()*, and you can use the four abstract methods as if they existed.

The class *SerialLoan* inherits *Loan*, and thus it is up to class *SerialLoan* to implement the four abstract methods so that *SerialLoan* is a concrete class. The class *SerialLoan* can implement the abstract methods, because it has the needed knowledge – a knowledge not known in the class *Loan*. The implementations of the abstract methods are trivial, but note that a *SerialLoan* is specifically a *Loan*.

The class *AnnuityLoan* is also easy to implement, and it consists simply of writing the above formulas in C#. Here you should note the method *Pow()*, which is a static method of the *Math* class, which raises an argument in a power.

Finally, there is class *Amortisation*, which can print a plan for a loan's amortization. Here you should note that the class has an instance variable of type *Loan*, which is initialized in the constructor. So you can have a variable whose type is an abstract class, since concrete objects of types respectively *SerialLoan* and *AnnuityLoan* also is of the type *Loan*. Please note that the class *Amortisation* know nothing about the specific loan type, but nevertheless one can write method *Print()*, since all methods are defined for the abstract type of *Loan*. The class *Amortisation* will also support and work on new types of loans as long as they inherit the class *Loan*.

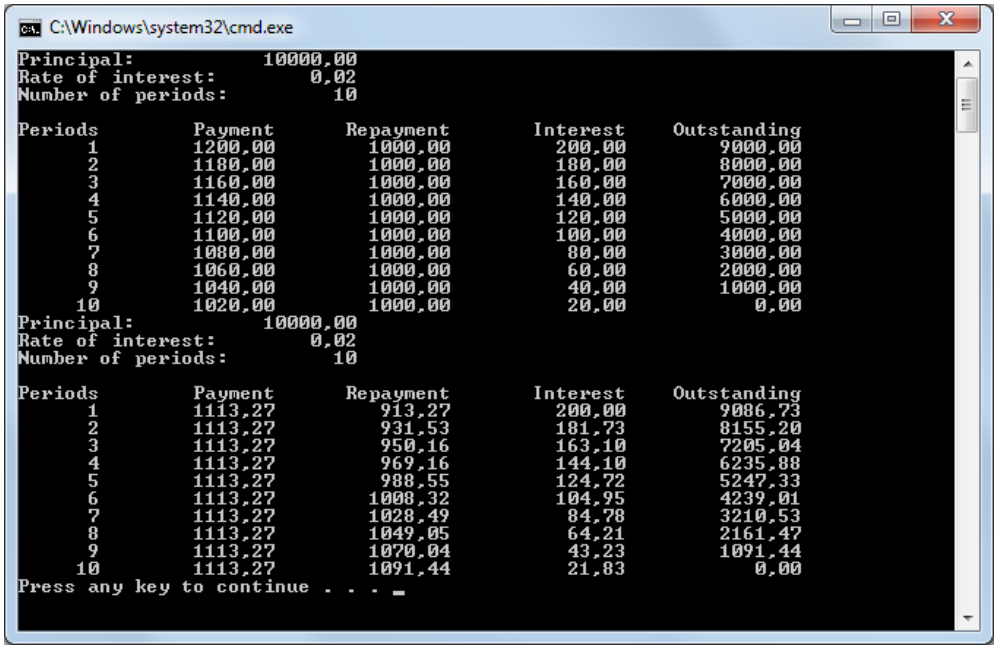
Download free eBooks at bookboon.com

Test

If you test the above classes with the following program:

```
class Program
{
   static void Main(string[] args)
   {
      Amortisation table1 = new Amortisation(new SerialLoan(10000, 0.02, 10));
      Amortisation table2 = new Amortisation(new AnnuityLoan(10000, 0.02, 10));
      table1.Print();
      table2.Print();
   }
}
```

you get the following result:



Comment

In the last examples, there typically have been a number of classes and in this case four classes in addition to the *Main* program. In all examples, the classes code have been in the same file, but as the examples become larger, there is not a viable option for the sake of clarity. Many choose to consistently place each a class in its own file and it is quite a sensible strategy, since it is a way for small source files, which are much easier to understand.

Comment

Note in particular that a class may be abstract, although it has not abstract methods. It can be used if you want to ensure that a class can not be instantiated.

Download free eBooks at bookboon.com

14 Interfaces

An abstract class is a type that can contain everything that a class can have. That is variables, constructors, methods, etc. Just may some of the methods be abstract, corresponding to those not yet been encoded – it is deferred to the concrete classes that have the requisite knowledge. In contrast, an interface is a type which can contain only abstract methods.

Exam33

Points again

The following interface defines a point:

```
public interface IPoint
{
   int X { get; set; }
   int Y { get; set; }
}
```

Basically it looks like a class, just stands there instead the word *interface*. An *interface* defines the methods and properties, and in this case two properties. One can think of an interface as a class that can only contain abstract methods. Note especially that in an interface has no method visibility – they are by default public.

Below is a class that implements the interface:

```
public class Point : IPoint
{
   private int x;
   private int y;

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

   public int X
   {
      get { return x; }
      set { x = value; }
   }

   public int Y
   {
      get { return y; }
      set { y = value; }
   }

   public override string ToString()
   {
      return string.Format("({0}, {1})", x, y);
   }
}
```


Download free eBooks at bookboon.com

Note that you are using the same syntax as for inheritance:

```
public class Point : IPoint
```

When a class implements an interface, the class must implement the methods and properties that are defined in the interface.

The following *Main()* method uses the interface:

```
static void Main(string[] args)
{
   IPoint p = new Point(2, 3);
   Console.WriteLine(p);
}
```

p is a *Point* object, but in the program it is known by its defining interface.

Exam30

Money

In this example I will write a program that creates a purse with bank notes. The purse must provide the following services available:

- one can place a bank note in the purse
- one can ask whether the purse is empty
- one can ask about the purse is full
- one can ask about the purse has (contains) a particular bank note
- one can take (pay with) a bank note with a specified value
- one can get to know, how much money is in the purse

When the purse is implemented, it must be tested by a program, which is cash in some money in the purse and then uses some of the notes.

How to

The first step is to define a bank note and I will work with Danish bank notes that have values 50, 100, 200, 500 and 1000 and thus 5 different notes.

Download free eBooks at bookboon.com

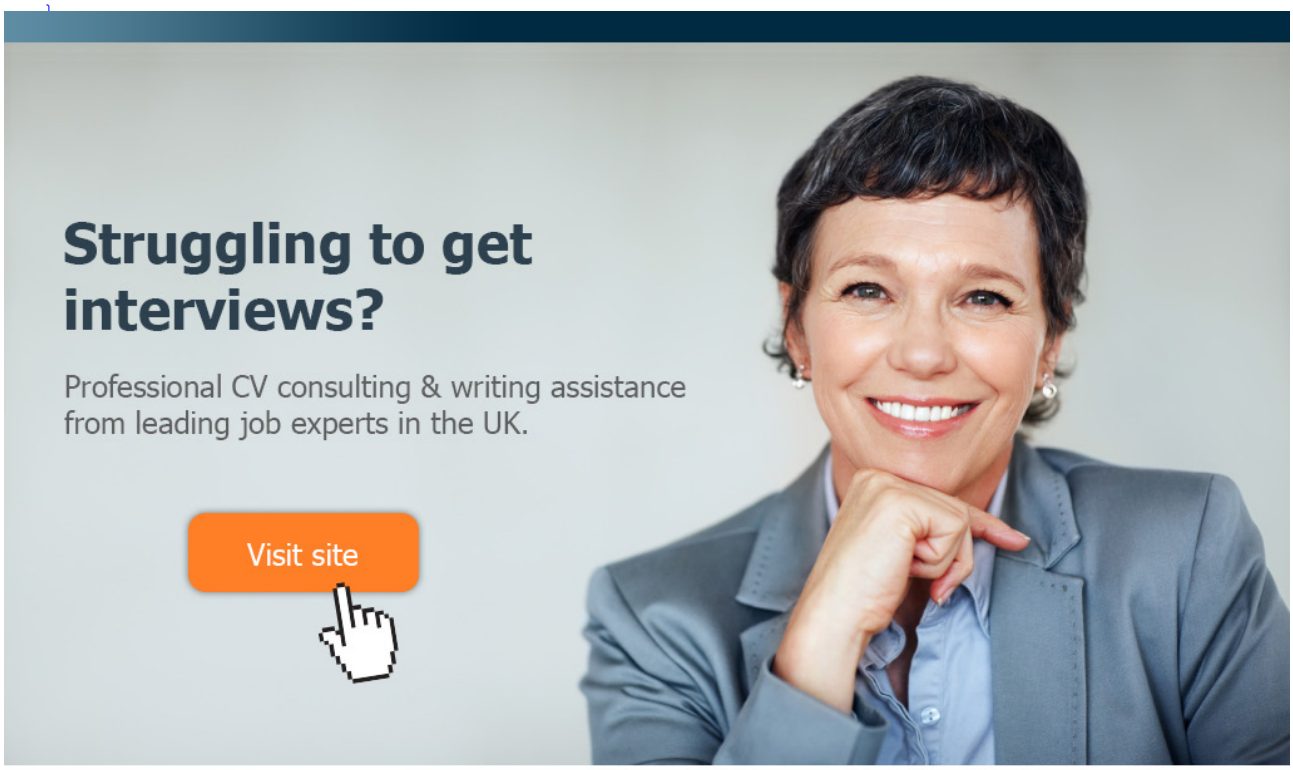
A bank note is a very simple concept which mainly is characterized by a value (one of the above), and can be defined by an interface:

```
public interface IBankNote
{
   int Value { get; }
}
```

which is merely a definition that says that a bank note – a *IBankNote* object – is something that has a get property, which returns an int.

I will then define an abstract class that implements the above interface and the methods that must be applied to all notes:

```
public abstract class BankNote : IBankNote
{
   public abstract int Value { get; }

   public override bool Equals(object obj)
   {
      if (!(obj is IBankNote)) return false;
      return ((IBankNote)obj).Value == Value;
   }

   public override int GetHashCode()
   {
      return Value;
```



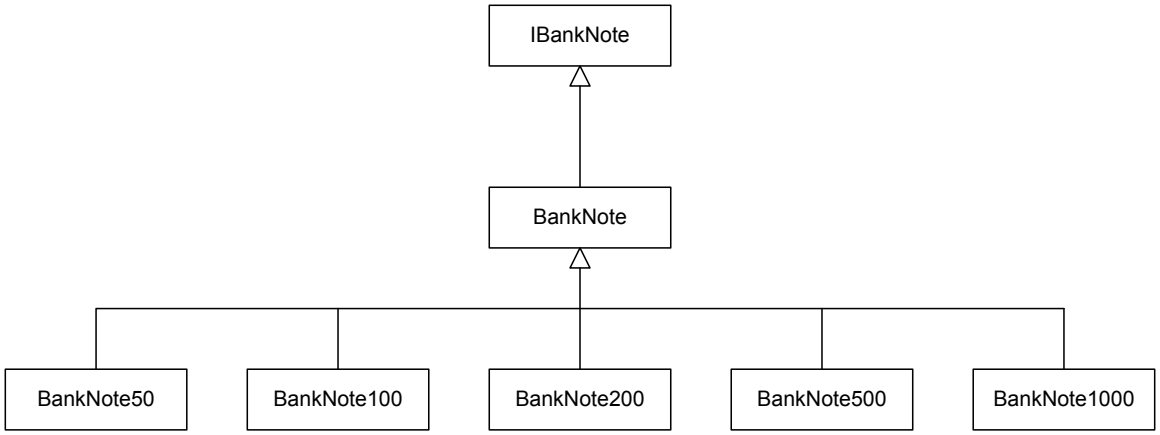
Download free eBooks at bookboon.com

```
   public override string ToString()
   {
      return Value + " Danish Crowns";
   }
}
```

With this class available, you can define concrete classes for the different notes, for example:

```
public class BankNote50 : BankNote
{
   public override int Value
   {
      get { return 50; }
   }
}
```

This means that there are defined the following class hierarchy:



Explanation

Before I look at the implementation of the purse I will make a few comments on the diagram above and what an interface is.

The interface *IBankNote* defines only a single read only property to return the value of a bank note. Note that it has no visibility. It is by default *public*, and we can't provide any visibility as it provides a translation error. An interface is a very simple concept, which only defines that a type has certain properties, but the interface does not implement this features.

You should be particularly aware that an interface is a type like a class, and as such you can have a variable whose type is an interface, but you can – naturally enough – not instantiate an object with that type. For example would the following be an illegal statement:

```
IBankNote note = new IBankNote();       // illegal, an interface can't be instantiated
```

As a last remark concerning interfaces, it is customary in C# to let the name of an interface start with a capital I, that is, for example *IBankNote* and it is advisable to keep to this practice.

Download free eBooks at bookboon.com

Interfaces must be implemented by classes and the class *BankNote* implements the interface *IBankNote*. Note that the syntax is the same as for inheritance:

```
public abstract class BankNote : IBankNote
```

that is a colon followed by the interface name. The class *BankNote* does not contain anything new, because it primarily overrides three methods from the class *Object*. Although it is not specified, the *BankNote* class inherits *Object*, and it is actually allowed (but unnecessary) to write:

```
public abstract class BankNote : Object, IBankNote
```

Since the class implements the interface *IBankNote*, it must implement the interface's properties and methods. It can't, because the class does not know what value the note should have, so instead it must make the property *Value* abstract. *BankNote* is then an abstract class.

Note also that the implementation of methods such as *ToString()* uses the property *Value*, and there is nothing wrong with that, although it has not yet been implemented.

Then finally there are the concrete classes, and they are extremely simple and do not require special explanation, but note that, for example a *BankNote50* is also a *BankNote* and thereby also an *IBankNote*.

The class *Purse*

The task was to implement a purse for notes where the purse should have 6 properties. Since this example deals with interfaces, I will define a purse in the form of an interface:

```
public interface IPurse
{
   bool Put(IBankNote note);
   bool IsEmpty();
   bool IsFull();
   bool Has(IBankNote note);
   IBankNote Pay(int value);
   int Value();
}
```

The interface defines that a purse must have six methods and hence the properties that a purse should have and what you can do with a purse, but the interface defines nothing about how the purse will be implemented. It requires a class that implements the interface:

```
public class Purse : IPurse
   {
   private IBankNote[] list;
   private int count;

   public Purse(int size)
   {
      list = new IBankNote[size];
   }
```


Download free eBooks at bookboon.com

```
public bool Put(IBankNote note)
{
   if (count >= list.Length) return false;
   list[count++] = note;
   return true;
}

public bool IsEmpty()
{
   return count == 0;
}

public bool IsFull()
{
   return count == list.Length;
}

public bool Has(IBankNote note)
{
   for (int i = 0; i < count; ++i) if (list[i].Equals(note)) return true;
   return false;
}

public IBankNote Pay(int value)
{
   for (int i = 0; i < count; ++i)
      if (list[i].Value == value)
      {
         IBankNote note = list[i];
         list[i] = list[--count];
         return note;
      }
```

Click on the ad to read more

Download free eBooks at bookboon.com

```
      return null;
   }

   public int Value()
   {
      int sum = 0;
      for (int i = 0; i < count; ++i) sum += list[i].Value;
      return sum;
   }
}
```

Explanation

Note first that the class implements the interface *IPurse*. Note then that there is defined an array with name *list* to the bank notes of the purse, and the type of the array is *IBankNote*, that is an interface. The array is created in the constructor that has a parameter that tells how many notes the purse must have room for. There is also a variable *count*, which keeps track of how many notes there are in the purse. After the purse is created it is empty.

Most of the methods are simple. The method *Put()* must leave a note in the purse, but it can only do it if there is room. Therefore it must test, if it is. If there is not space, the method returns false without doing anything. Otherwise, it put the note in the purse and returns true. The two methods *IsEmpty()* and *IsFull()* is quite trivial. The method *Has()* go through the notes in the purse with a *for* loop to check if you have the bank note, which is queried. Note that this is a requirement that the note class implements the method *Equals()* correctly with value semantic. It's something that the abstract class *BankNote* solves.

The method where there is most to note is the method *Pay()*. Basically it consists of a loop that runs through the notes in the purse to find a note with the correct value. If you find a note, there are two things to do: the note must be removed from the purse, and the method must return the note. The first thing is solved by putting the note on the last place onto the place where the note should be removed. Note that this requires that you first save a copy of the note to be returned. Also note that the variable *count* is counted down with the one. If the purse does not have a bank note with the desired value, there is a problem because that the method should return something. It has been solved by letting the method returns a *null* reference. It is a solution that can be discussed much, but conversely a solution that is widely used, and at least gives the user the ability to test whether the purses had the desired note.

It is important to note that both the interface *IPurse* and the implementation *Purse* know nothing about the actual banknote classes, but only knows the interface *IBankNote*. It means that you with no problems can add new banknotes classes and without the need to modify the code for *Purse*.

The last method requires no special comment, and consists merely of a simple pass through the notes to calculate the total value.

Download free eBooks at bookboon.com

The program

There remains then the program itself, which should create a purse, put banknotes into it and then spends some money:

```
class Program
{
   static Random rand = new Random();

   static void Main(string[] args)
   {
      IPurse purse = new Purse(10);
      Init(purse);
      Console.WriteLine(purse.Value());
      for (int i = 0; i < 20; ++i) Buy(purse, Create());
      Console.WriteLine(purse.Value());
   }

   static void Buy(IPurse purse, IBankNote note)
   {
      if (purse.Has(note))
      {
         purse.Pay(note.Value);
         Console.WriteLine("Bought for {0} crones", note.Value);
      }
      else
         Console.WriteLine("Does not have a " + note);
      }

   static void Init(IPurse purse)
   {
      while (!purse.IsFull()) purse.Put(Create());
   }

   static IBankNote Create()
   {
      switch (rand.Next(15))
      {
         case 0: return new BankNote1000();
         case 1:
         case 2: return new BankNote500();
         case 3:
         case 4:
         case 5: return new BankNote200();
         case 6:
         case 7:
         case 8:
         case 9: return new BankNote100();
         default: return new BankNote50();
      }
   }
}
```


Download free eBooks at bookboon.com

Explanation

Note first the method *Init()*, which fills the purse with bank notes.

Notes are created by the method *Create()* which creates a random bank note. Note that this is the only place in the entire program, where the concrete bank note classes appear – all else is a bank note only known as the interface *IBankNote*. In fact, one could usefully move the code for the method *Create()* to its own factory class, so you got an even better separation of the program and the specific bank note classes.

The method *Buy()* simulates that you pay with a particular banknote. This method asks where the purse has a particular note, and if it's true the method "buy for the amount". Otherwise it may say "sorry". Please note that this method only knows the purse through the interface *IPurse*, and the method will work even if a purse was implemented in a different way, as long as it implements the interface.

The program is as follows:

```
static void Main(string[] args)
{
   Pung pung = new ArrPung(10);
   Init(pung);
   Console.WriteLine(pung.Ialt());
   for (int i = 0; i < 20; ++i) Koeb(pung, Create());
   Console.WriteLine(pung.Ialt());
}
```

and below is an example of a test of the program.

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

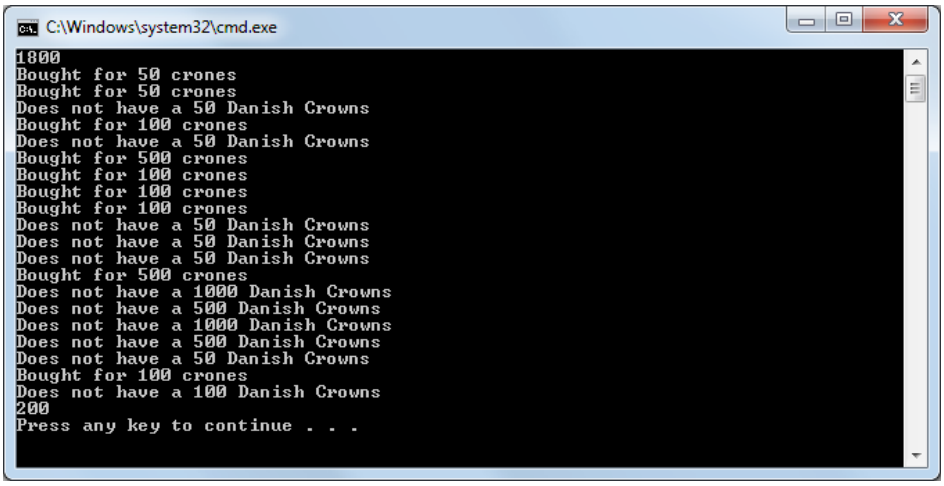
We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

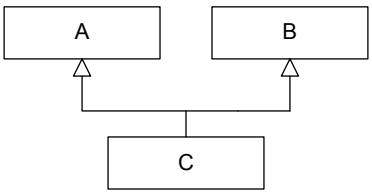


Comment

There are basically two objectives for interfaces. In the first place this is a good programming principle is to define a subject in the form of an interface, that is, for example *IPurse* above. It is said that one should program to an interface. In this way, the definition and implementation is separated, and the important thing is that the code that uses the subject only knows it through the interface and thus is completely independent of the implementation. This ensures that you can change the implementation without the need to modify the code that use the subject. Programming to an interface may therefore help to ensure that you get a code that is easier to modify. The above program is "almost" independent of the implementation of the interface *IPurse* and there is also only a place where the implementation is clear, namely, where the *Main()* creates the purse:

```
IPurse pung = new Purse(10);
```

Another objective of an interface is to allow for on the design level to work with multiple inheritance. As mentioned above, a class can only have a single base class – a class inheritance only from one class. It is called sometimes for linear inheritance. You can't have a design as shown below:



where a class C inherits both the class A and the class B. There are several reasons why it is not possible. Firstly, there are difficulties in implementing the concept in a programming language in a meaningful way, and secondly, one can discuss whether it is an appropriate design, as it expresses that C is both an A and B. There are examples where it is reasonable, and in C# a class can inherit a single class, but implement all the interfaces, it may have wanted or needed. You'll have seen many examples of that later.

Download free eBooks at bookboon.com

15 Static members

I would like once again to look at static members in a class. Both variables and methods can be static, and indeed can a class be static.

When you create an object of a class, you create simultaneously the data elements as simple variables and other objects that the class's instance variables define. Each object of the class has its own copy of all instance variables, and if an object changes the value of an instance variable it only concerns the object itself.

Things are different with static variables as a static variable is created only for the first time an object of the class is created. This means that all objects created from the same class share a static variable, or said differently that a static variable is not tied to a particular object.

A good example of using a static variable is a random number generator – an object of type *Random* – as in the classes *Coin* and *Dice*. Here it is important that all objects (for example all *Dice* objects) are using the same random number generator. The static random number generator is initialized in the declaration of the static variable:

```
private static Random rand = new Random();
```

It is not always possible or desirable, and one can instead initialize static variables in a static constructor that has no other purpose than to initialize static variables. For example you could write the class *Dice*, as follows:

```
public class Dice
{
   private static Random rand;
   private int oejne;

   static Dice()
   {
      rand = new Random();
   }
```

There is no justification for it in this case, but a static constructor is executed the first time a program creates an object of the class. You can't override a static constructor – there are no options to transfer parameters.

Also, methods can be static. If you have a static method, you can use it without having an object of the class, and the method is referenced by setting the class name before method name.

Download free eBooks at bookboon.com

Exam34

StringBuilder

In this example I will show a class that contains only static methods. The class will consist of methods to manipulate the strings, and is a class which may be useful in practice. I should also mention the class *StringBuilder*, a class that can give increased efficiency in situations with many operations on strings. Finally, the example has a test program that tests the class.

How to

The starting point is a class *Str* with 4 static methods:

```
public static class Str
   {
   public static string Cut(string text, int width)
   {
      if (text.Length > width) return text.Substring(0, width);
      return text;
   }

   public static string FillRight(string text, int width, char ch)
   {
      StringBuilder builder = new StringBuilder(text);
      while (builder.Length < width) builder.Append(ch);
      return builder.ToString();
   }
```

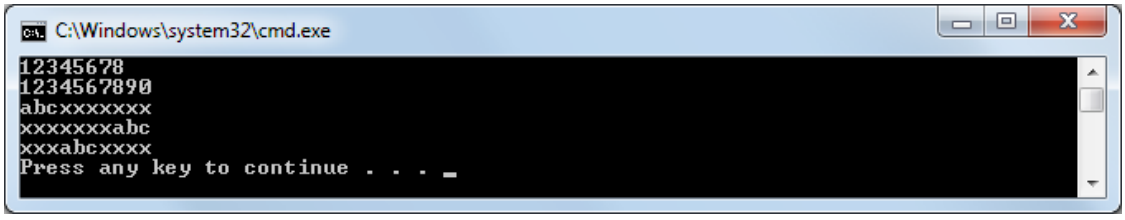
Click on the ad to read more
Download free eBooks at bookboon.com

```
   public static string FillLeft(string text, int width, char ch)
   {
      if (text.Length >= width) return text;
      StringBuilder builder = new StringBuilder(width - text.Length);
      for (int i = 0; i < builder.Capacity; ++i) builder.Append(ch);
      return builder.ToString() + text;
   }

   public static string FillCenter(string text, int width, char ch)
   {
      return FillRight(FillLeft(text, text.Length + (width - text.Length) / 2, ch),
      width, ch);
   }
}
```

In this case, the class *Str* is located in its own file. This provides better opportunities to use the class in other programs.

Below is a test program:

```
class Program
{
   static void Main(string[] args)
   {
      Test1();
      Test2();
   }

   static void Test1()
   {
      Console.WriteLine(Str.Cut("1234567890", 8));
      Console.WriteLine(Str.Cut("1234567890", 12));
      }

   static void Test2()
   {
      Console.WriteLine(Str.FillRight("abc", 10, 'x'));
      Console.WriteLine(Str.FillLeft("abc", 10, 'x'));
      Console.WriteLine(Str.FillCenter("abc", 10, 'x'));
   }
}
```



Download free eBooks at bookboon.com

Explanation

The first method cut of the string, so that it has a maximum width (number of characters):

```
public static string Cut(string text, int width)
{
   if (text.Length > width) return text.Substring(0, width);
   return text;
}
```

It is a static method (in fact a very simple but useful method), and it may, for example be used as follows:

```
Console.WriteLine(Str.Cut("1234567890", 8));
```

You should note that the method referenced by typing class name *Str* in front.

Basically, a static method can do the same as other methods, and it is important that it can be used without an object. In turn, a static method can't refer to instance variables, but there are also many examples of methods, that does not. A good example is the *Math* class, which contains a number of mathematical functions. They are all implemented as static methods.

The second method is a method that extends a string to a minimum width, such that it is filled out to the right with a specific character *ch*. The method uses a *StringBuilder*, but it could be written differently:

```
public static string FillRight(string text, int width, char ch)
{
   while (text.Length < width) text += ch;
   return text;
}
```

A string object can't change state – you can't change the content of a string. If you look at the expression

```
text += ch;
```

it means to create a *string* object that contains the previous content of *text* and expanded it with value *ch*. That means, the creation of an object on the heap, and the old string have to be copied to it. In most cases it is not essential, but if there is to be added number of filler characters corresponding to the loop is repeated many times, it may affect the performance and be detectable. It is here the type *StringBuilder* comes into the picture. There is a kind of buffer for characters that can expand dynamically and when needed. The statement

```
StringBuilder builder = new StringBuilder(text);
```

creates a new *StringBuilder* with the content of *text*. The second loop adds the characters until the builder contains the desired number of characters, and note that it automatically expands as needed.

Download free eBooks at bookboon.com

It's hard to say exactly when to use a *StringBuilder*, but if there are more than 10 extensions you should consider whether it is worthwhile to take this class in use.

The third method is similar to *FillRight()*, but it fill character in from the left. It also uses a *StringBuilder*, but the parameter to the builder's constructor is this time is a number: the number of fill characters to be used. This means that the builder from the start can accommodate the required number of characters, and thus should not be expanded. Note also how, in the loop there are used a property *Capacity* to determine how many characters to be added.

If one considers the class *Str*, then all the methods are static. If so, you can also define the class as *static*:

```
public static class Str
{
```

If so, it is not possible to create objects of the type *Str*, and also it gives the no opinion in this case.

Comment

When you in Visual Studio creates a new console application it automatically creates a class with a *Main()* method. It is a *static* method and it must be, as it must be called by the runtime system without having an object. In most testing programs the class with *Main()* had only static methods, for example.

```
static void Test2()
{
   Console.WriteLine(Str.FillRight("abc", 10, 'x'));
   Console.WriteLine(Str.FillLeft("abc", 10, 'x'));
   Console.WriteLine(Str.FillCenter("abc", 10, 'x'));
}
```

and the method is usually called from *Main()*. When you in such situations has no object, the methods must be static, and that is why the methods in the *Main()* class has always been static.

Download free eBooks at bookboon.com

16 More about arrays

I have previously defined an array as a number of elements of a particular type that can be referenced via a common name. The picture of an array is a structure like the following

where each box has room for an element of the arrays type. The individual elements can be referenced using the array name and an index that always starts from 0. The type can be anything, and in the class *Cup* it was a *Dice*, while in the class *Purse* it was an *IBankNote*. The only thing to note is that if the type is a value type, the boxes directly contains the value that is attached to the individual indices, but if the type is a reference type, the boxes contains only references to the objects that are linked to the individual objects. It is rare that it means so much in practice, but you should be aware that if you write something like the following

```
Dice[] t = new Dice[5];
```

then there is created an array, but there it is not yet filled with *Dice* objects. The array is empty corresponding to each position contains *null*. There is not yet created any *Dice* objects.

This e-book
is made with
SetaPDF

SETASIGN

PDF components for PHP developers
www.setasign.com

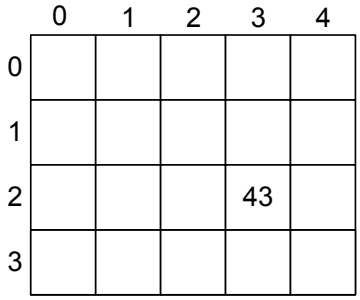
Click on the ad to read more

Download free eBooks at bookboon.com

An array as above is a 1-dimensional array. One can also work with arrays of multiple dimensions. For example is a 2-dimensional array a structure organized into rows and columns, and each element can be referenced using the array name and an index pair. For example you can define a two-dimensional array of elements of the type *int* having 4 rows and five columns in the following manner:

```
int[,] t = new int[4, 5];
t[2, 3] = 43;
```

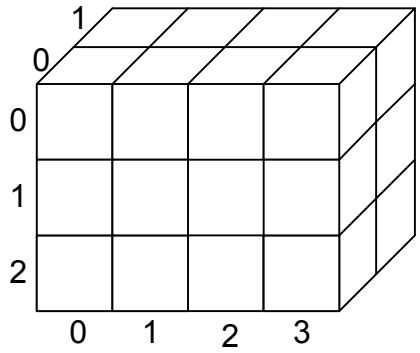
This can be illustrated in the figure below. Please note that as with a 1-dimensional array indices start with 0 – for both rows and columns.



It is seldom you are using arrays with more than two dimensions, but there is no upper limit to the number of dimensions, but for us humans it is difficult to give the array a geometric interpretation. For example you can define a 3-dimensional array as:

```
int[,,] t = new int[3, 4, 2];
```

This can be illustrated as a cube (or more 2-dimensional arrays, lying behind each other):



The individual elements referenced by three indices, for example.

```
t[1, 2, 1] = 53;
```

Then it is easy to guess the syntax of how to define arrays of dimension greater than 3.

Download free eBooks at bookboon.com

Exam35

Multi-dimensional arrays

In this example I will show some examples of multidimensional arrays.

How to

The examples are as follows:

```
static void Test1()
{
   char[,] t = new char[5, 4];
   char ch = 'A';
   for (int i = 0; i < t.GetLength(0); ++i)
      for (int j = 0; j < t.GetLength(1); ++j) t[i, j] = ch++;
   Print(t);
}

static void Test2()
{
   char[,,] t = new char[3, 5, 4];
   char ch = 'A';
   for (int i = 0; i < t.GetLength(0); ++i)
      for (int j = 0; j < t.GetLength(1); ++j)
         for (int k = 0; k < t.GetLength(2); ++k) t[i, j, k] = ch++;
   Print(t);
}

static void Test3()
{
   int[,] t = { { 2, 3, 5, 7 }, { 11, 13, 17, 19 }, { 23, 29, 31, 37 } };
   Print(t);
}

static void Test4()
{
   char[][] t = new char[4][];
   t[0] = new char[3];
   t[1] = new char[5];
   t[2] = new char[2];
   t[3] = new char[7];
   char ch = 'A';
   for (int i = 0; i < t.Length; ++i)
      for (int j = 0; j < t[i].Length; ++j) t[i][j] = ch++;
   Print(t);
}

static void Test5()
{
   int[] t = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
   foreach (int n in t) Console.WriteLine(n);
}

static void Test6()
{
   int[] t = { 23, 7, 5, 11, 3, 17, 29, 2, 19, 13 };
   Array.Sort(t);
   foreach (int n in t) Console.WriteLine(n);
}
```


Download free eBooks at bookboon.com

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

static void Print(char[,,] t)
{
   for (int i = 0; i < t.GetLength(0); ++i)
   {
      for (int j = 0; j < t.GetLength(1); ++j)
      {
         for (int k = 0; k < t.GetLength(2); ++k) Console.Write("{0} ", t[i, j, k]);
         Console.WriteLine();
      }
      Console.WriteLine();
   }
}

static void Print(int[,] t)
{
   for (int i = 0; i < t.GetLength(0); ++i)
   {
      for (int j = 0; j < t.GetLength(1); ++j) Console.Write("{0, 3:D} ", t[i, j]);
      Console.WriteLine();
   }
}
```

Click on the ad to read more

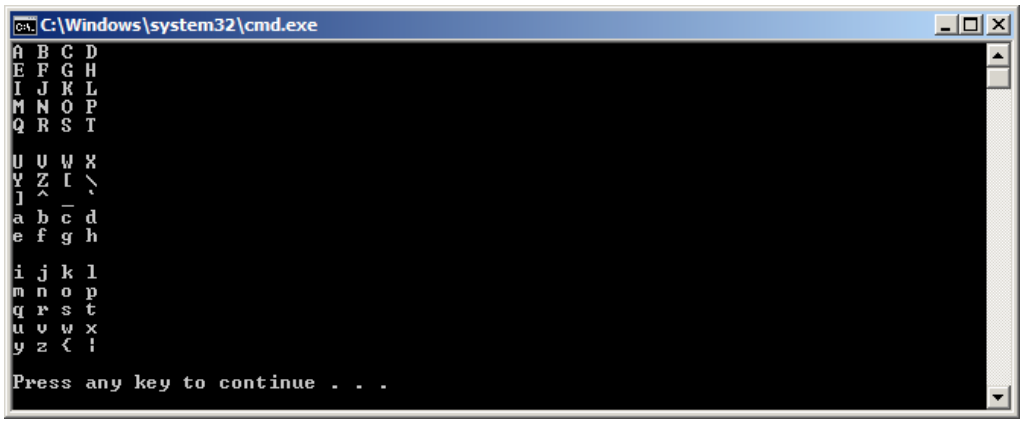
Download free eBooks at bookboon.com

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

Explanation

The first example creates a 2-dimensional array of the type *char* and prints it. You should particularly note how one refers to the number of elements in each dimension with the method *GetLength()*, where the parameter indicates the dimension that you refer.

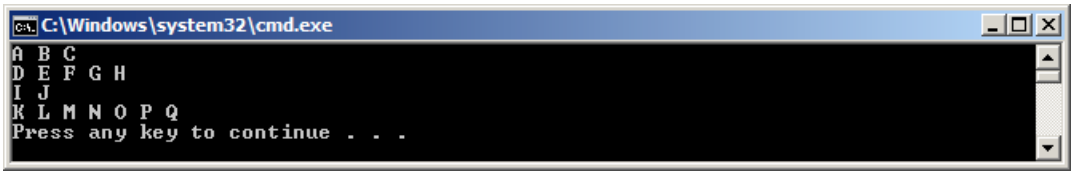
The next example is similar, but here is instead talking about a 3-dimensional array. You should be especially aware of how to create a 3-dimensional array – here with 5 rows, 4 columns and 3 layers. Note also how to refer to an element using three indices.



It is also possible to initialize a multidimensional array in the declaration using a list. The third example shows how to create a 2-dimensional array consisting of 3 rows and 4 columns.

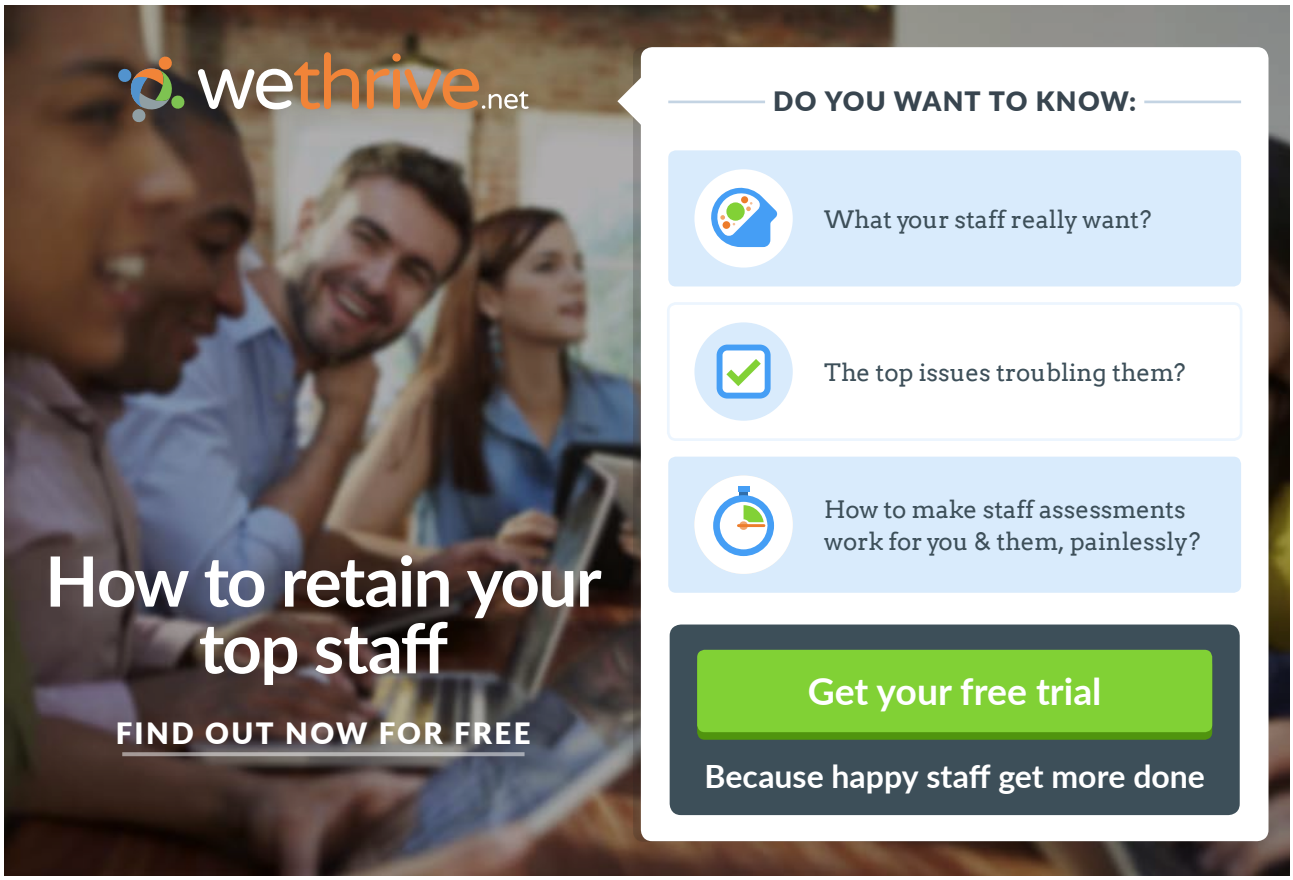
The type of an array can be anything and thus specially also another array. This makes it possible to define the arrays, where each row has a different number of elements. The fourth example creates an array of four rows where the numbers of elements per array are respectively 3, 5, 2 and 7:

Download free eBooks at bookboon.com



As a further remark concerning arrays, let me mention another loop construction. Indeed, it has not specifically to do with arrays, but it may be useful in the context of arrays. The fifth example defines a generally 1-dimensional array of 10 elements. The array is printed on the screen pass it with a loop, but this time using a *foreach* loop. The syntax is simply to define a variable of the same type as the array, and this variable will then run through all array elements. The advantage of *foreach* rather than a simple *for* statement is simply that it may be more readable.

As a last remark concerning arrays, I will mention the class *Array*, which is the base class for any array, and thus also provides a range of methods and properties available. Specific the class provides multiple static methods available to manipulate arrays. It is worthwhile to investigate this class and see what there is to work with. The last example defines an array and sorts it.



Click on the ad to read more

Download free eBooks at bookboon.com

17 Types

C# has several categories of types, including class types and interface types, but also simple types like *int* and *double*. Characteristics of these types and how variables / objects of these types are allocated, are, as mentioned above, generally divided into value types and reference types.

A variable of a value type allocates space on the stack to its data, and one can say that the variable has its own copy of or contains the data stored in it. The variable is automatically destroyed (removed from the stack) when the program finishes the block where the variable is declared. A variable of a value type uses thus the number of bytes of the stack, which the type indicates.

A variable of a reference type does not contain data directly, but contains instead a reference to data that is allocated on the heap. This means that one can have several variables that relate to the same data. This means also that a reference variable always has the same size of the stack that is the size of a reference (4 bytes). A reference can specifically have the value *null*, which means it does not refer to anything. Reference types can be classified into class types, arrays, and interface types.

The next sections deals more on the types and value types primarily, but also dealt with generic types and exception handling, but I will start with a few more remarks on the simple types.

The simple or built-in types are, for example *int*, *char*, *double*, etc. Each type has a name that is a reserved word, but these names are really just an alias for a similar *struct* in the *System* namespace. This means that there are also associated methods to the simple types. For example is *int* an alias for the *System.Int32*, which is a 4 byte integer.

The following table shows the simple types in the namespace System:

C# alias	Type i System	Betydning
sbyte	SByte	8 bit signed integer
byte	Byte	8 bit unsigned integer
short	Int16	16 bit signed integer
ushort	UInt16	16 bit unsigned integer
int	Int32	32 bit signed integer
uint	UInt32	32 bit unsigned integer
long	Int64	64 bit signed integer
ulong	UInt64	64 bit unsigned integer
char	Char	16 bit Unicode character
float	Single	32 floating-point number
double	Double	64 floating-point number
bool	Boolean	8 bit that is *true* or *false*
decimal	Decimal	96 bit decimal number with 28 significant digits

Download free eBooks at bookboon.com

The simple types are all derived from the type *ValueType* (which is a *struct*, and the term *struct* is explained below) and which is in turn derived from the *Object*. The types can basically be divided as

- *char*
- *bool*
- types to integers (*sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*)
- types to decimal numbers (*float*, *double*, *decimal*)

The type char is simple, since it represents the individual characters as a 16-bit numeric code in the *unicode* system. There are tables that show which encodes each character has. Below is shown a method that prints the characters with codes from 32 up to and including 255:

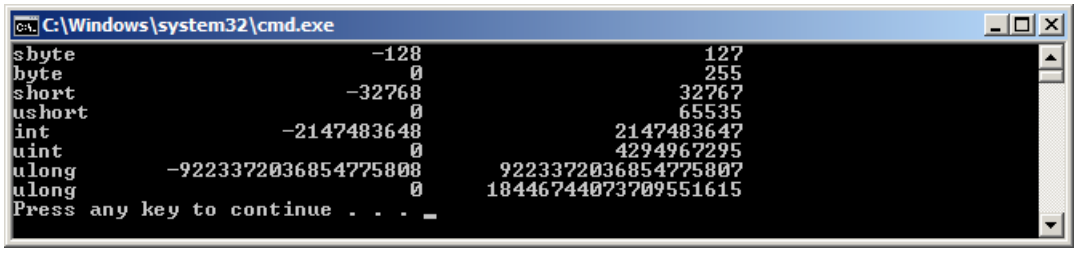
```
static void Test1()
{
   for (char c = ' '; c <= 255; ++c) Console.WriteLine("{0}{1, 4:D}", c, (int)c);
}
```

Note especially the comparison in the *for* statement where a *char* is compared with an *int*, and that the + + operator also makes sense for a *char*. By comparison, the characters code number is compared, and + + works because it is the code that counted.

To the type *bool* can't be made more comments, but the number types are worth a closer look. A *sbyte* fill 1 byte, and thus 8 bits. It can therefore represent any integer that binary can be written with 8 bits, which are the numbers from -128 to 127. The slightly oblique zone has to do with the internal representation where a negative integer is represented by its 2-complement. If instead you have a *byte*, its values range from 0 to 255. The difference between a *sbyte* and a *byte* is thus simply a parallel shift of the range which is representative of the type in which the one is a symmetrical range of 0, while the other is the non-negative integers starting with 0. The same is true for the types *short*, *int* and *long* the difference is only the size of the range of integers, which are available. The following method prints these intervals for all 8 integer types:

```
static void Test2()
{
   Console.WriteLine("sbyte  {0, 25:D} {1, 25:D}",  sbyte.MinValue, sbyte.MaxValue);
   Console.WriteLine("byte   {0, 25:D} {1, 25:D}",  byte.MinValue, byte.MaxValue);
   Console.WriteLine("short  {0, 25:D} {1, 25:D}",  short.MinValue, short.MaxValue);
   Console.WriteLine("ushort {0, 25:D} {1, 25:D}",  ushort.MinValue,ushort. MaxValue);
   Console.WriteLine("int    {0, 25:D} {1, 25:D}",  int.MinValue, int.MaxValue);
   Console.WriteLine("uint   {0, 25:D} {1, 25:D}",  uint.MinValue, uint.MaxValue);
   Console.WriteLine("ulong  {0, 25:D} {1, 25:D}",  long.MinValue, long.MaxValue);
   Console.WriteLine("ulong  {0, 25:D} {1, 25:D}",  ulong.MinValue, ulong.MaxValue);
}
```


Download free eBooks at bookboon.com



You can't just copy an integer of one type into another type without making a type cast, but the general rule is that you can copy a smaller type into a larger type. If you try to translate the following code

```
static void Test3()
{
   sbyte b1 = 2;
   byte b2 = b1;
   int n = 3;
   short s = n;
   long t = n;
}
```

you will get two translation errors. The statement

```
byte b2 = b1;
```

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

gives an error because you try to copy an integer which may be negative in a variable that can only contain no negative numbers. Similarly, the statement

```
short s = n;
```

cause an error when trying to copy 4 bytes into 2 bytes. In both cases the problem is solved with an explicit type cast:

```
static void Test3()
{
   sbyte b1 = 2;
   byte b2 = (byte)b1;
   int n = 3;
   short s = (short)n;
   long t = n;
}
```

As a last remark concerning integers, I will mention the possibility of statements that with integers as hexadecimal digits:

```
int a = 0x1abc23;
```

where 0x tells the compiler that the value should be interpreted as a hexadecimal number.

Back there are the decimal numbers that are represented by the types *float*, *double* and *decimal*. The first two is so-called floating-point numbers, while the latter is a decimal number. The difference is that the *float* and *double* meets a very large interval with fewer significant digits, while the latter supports many significant digits and return a more limited range. Common to the three types is that they represent only a portion of data within the intervals that they span, simply because that every interval contains infinitely many real numbers, and an infinite amount can't be represented with a finite number of bits. It is therefore important to realize that when working with decimal numbers, many of the results are rounded off and approximates the values, a relationship which is also known from an ordinary calculator. Especially in the context of comparisons it is important to be aware of this fact.

The difference between *float* and *double* are, how big a range, they span. The internal representation comprises of a sign, a mantissa with the digits and an exponent. If for example you calculates 3.14^{50}

```
Console.WriteLine(Math.Pow(3.14, 50));
```

you get the result

Download free eBooks at bookboon.com

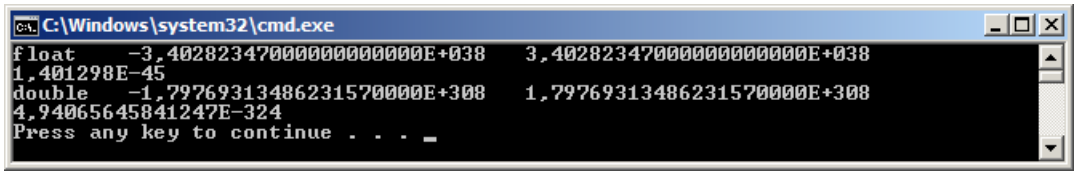
that means $7.02234890660125*10^{24}$. This means that a *double* works with approximate 15 significant digits. The above is thus a rounded result.

Note that you can also use that notation for a constant, for example

```
double x = 1234.678E+20;
```

The types *float* and *double* defines several constants and some of them are used in the following method:

```
static void Test4()
{
   Console.WriteLine(
      "float {0, 30:E20}{1, 30:E20}", float.MinValue, float.MaxValue);
   Console.WriteLine(float.Epsilon);
   Console.WriteLine(
      "double {0, 30:E20}{1, 30:E20}", double.MinValue, double.MaxValue);
   Console.WriteLine(double.Epsilon);
}
```



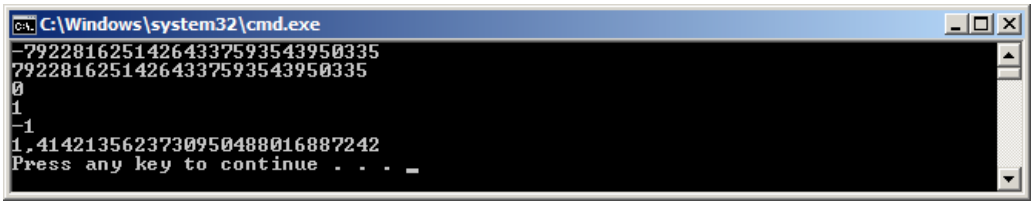
Here you can see the intervals the two types of spans, but you should also notice the constant *Epsilon*, which indicates the smallest positive number. Values are also available which do not represent numbers:

- `double.PositiveInfinity` representing the plus infinity
- `double.NegativeInfinity` representing minus infinity
- `double.NaN` that means a value that does not represent a number

The type decimal spans a smaller range, but in return for up to 29 significant digits:

```
static void Test5()
{
   Console.WriteLine(decimal.MinValue);
   Console.WriteLine(decimal.MaxValue);
   Console.WriteLine(decimal.Zero);
   Console.WriteLine(decimal.One);
   Console.WriteLine(decimal.MinusOne);
   decimal x = 2;
   Console.WriteLine(Sqrt(x));
}
```

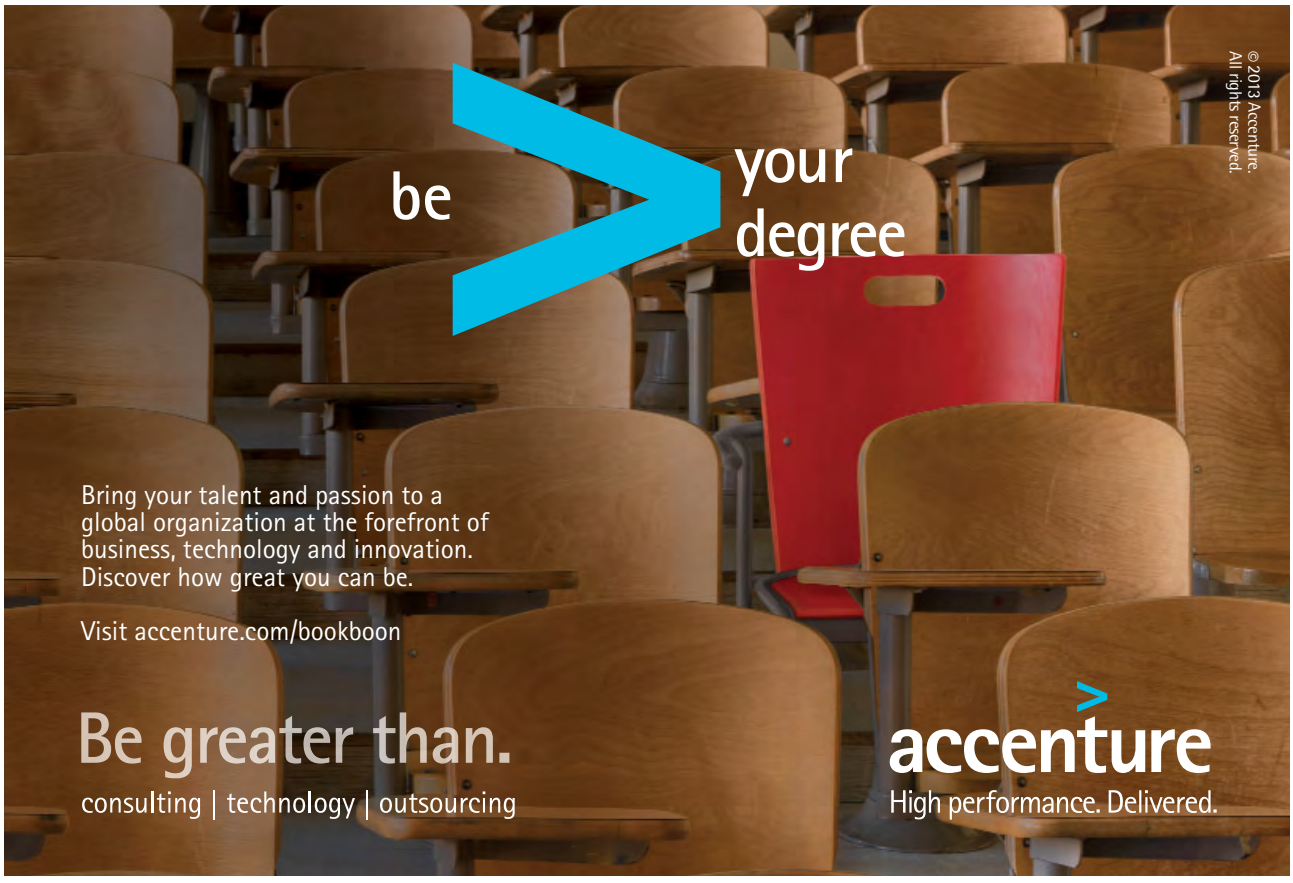

Download free eBooks at bookboon.com



Note that the type defines several constants. Also note the last statement that prints the square root of 2. The square root function in the class *Math* is not implemented for variables of type *decimal* then if you want to determine the square root of a *decimal* as a *decimal*, you have to write the function:

```
static decimal Sqrt(decimal x)
{
   decimal y = x;
   while (true)
   {
      decimal v = y + x / y;
      decimal z = v / 2;
      if (Math.Abs(y - z) <= decimal.Zero) break;
      y = z;
   }
   return y;
}
```

In general, the simple types are *unchecked*. That is, that they are not tested for overflow if a value is too large. If, for example you performs the following method



Click on the ad to read more

Download free eBooks at bookboon.com

```
static void Test6()
{
   int a = 1234567;
   int b = 1234567;
   Console.WriteLine(a * b);
}
```

you get the following result which is obviously not correct (the result can't be within the range as an *int* span):

When it is so, it is because it takes time to test for overflow, and if it should happen for all calculations in a program, it could reduce the program's effectiveness. Instead they have left it to the programmer to deal with situations that can cause overflow. This can be done as follows:

```
static void Test6()
{
   checked
   {
      int a = 1234567;
      int b = 1234567;
      Console.WriteLine(a * b);
   }
}
```

If you execute the method now, the program will stop with an error message – an exception. It is also possible to set an option to the compiler that all code should be checked. If this is done, there is also an *unchecked*, which can be used to select the blocks of which one does not wish to be *checked*.

In most cases, the difference between value types and reference types are not particularly important, but a variable of a value type and therefore especially also a variable of a simple type will always have a value. This is in contrast to a variable of a reference type, which may be *null*, and means that the variable do not have value. Sometimes it is useful also to operate with variables of value types that do not have a value, and therefore there are some special value types that can be *null*. The syntax is simple and you just need to write a ? after the type, that is, for example

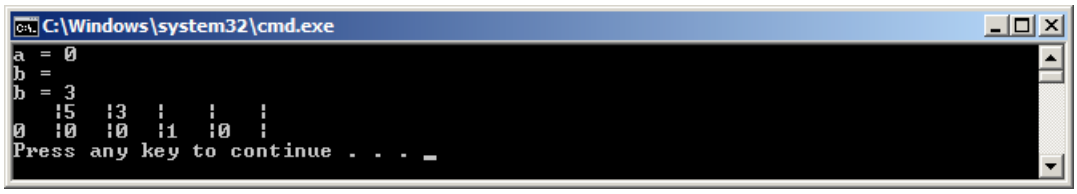
```
long? n;
double? x;
```

Such types are said to be *nullable*. It's simply means that the type is extended with a possibility that it may be *null*. Consider the following code:

Download free eBooks at bookboon.com

```
static void Test7()
{
   int a = 0;
   int? b = null;
   Console.WriteLine("a = " + a);
   Console.WriteLine("b = " + b);
   b = 3;
   Console.WriteLine("b = " + b);
   int?[] t = new int?[5];
   for (int i = 0; i < t.Length; ++i) t[i] = Number();
   Print(t);
   for (int i = 0; i < t.Length; ++i) t[i] = Number() ?? 0;
   Print(t);
}

static void Print(int?[] t)
{
   foreach (int? n in t) Console.Write("{0, -3}|", n);
   Console.WriteLine();
}

static int? Number()
   {
   if (rand.Next(2) == 1) return rand.Next(1, 10);
   return null;
}
```

First the method declares two variables: A common *int* and a *nullable int*. The first time *b* is printed it is *null*, but the second time it has been given a value. It serves to illustrate that apart from a *nullable* variable can be *null* it is used just like any other variables. *int?* is a type as all other types, and therefore one can specially create an array with elements of this type. Similarly the type can be used both as a return type and as a parameter. The method *Number()* returns a value that is either an integer or is *null*, and the method *Print()* prints a *nullable* array. Note also the operator ??, which means that the value is the expression after ?? if the value before is null.



All of the above test methods are found in the example Exam35.

Download free eBooks at bookboon.com

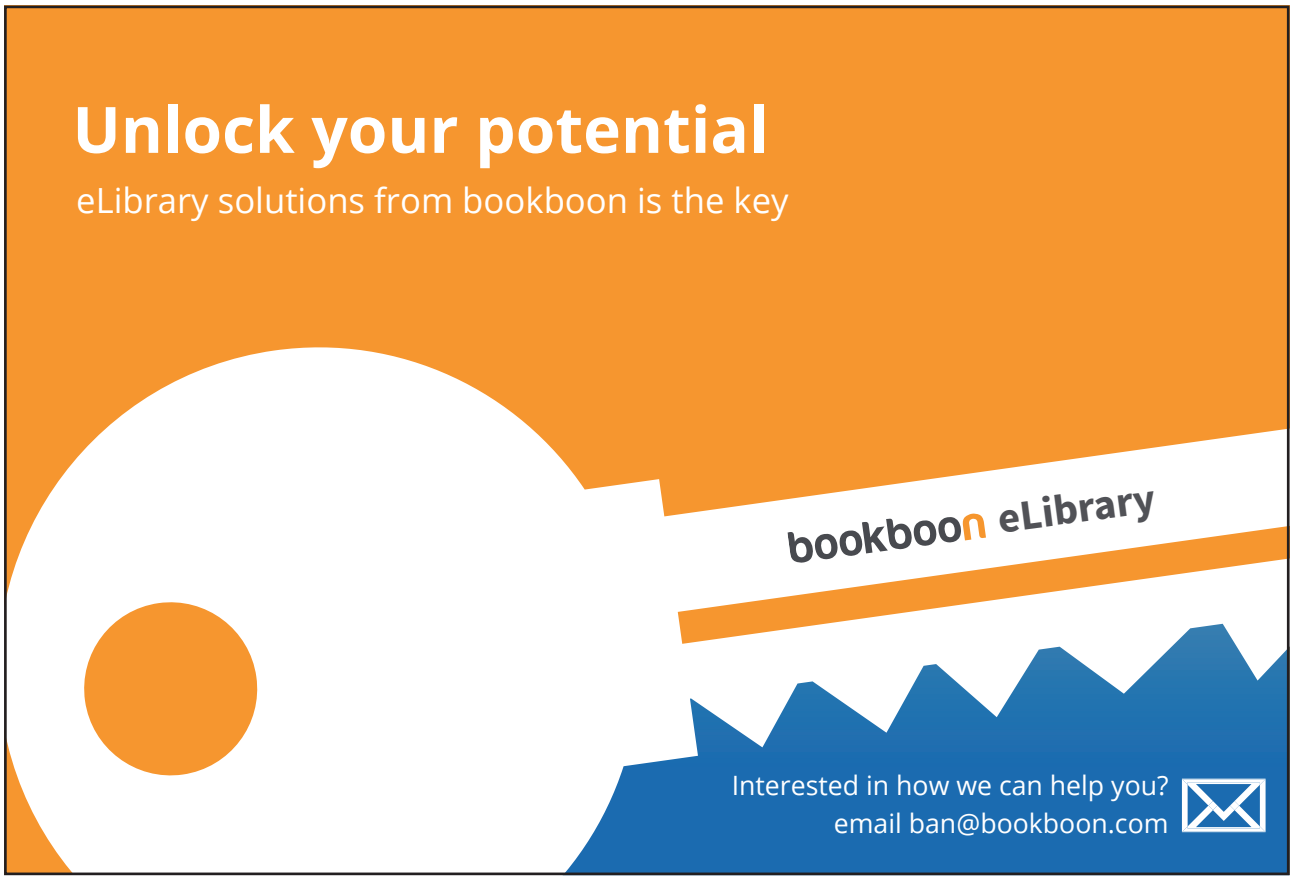
18 Enum

The value type *enum* assigns names to constants of one of the types *byte*, *short*, *int* and *long*. As default are assigned constants values from 0 onwards, but you can also explicitly assign a constant a value. An enumeration is declared in the following manner:

```
public enum Ugedag : byte
{
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday,
   Sunday,
   Error = 10
}
```

which declared 8 constants. The type is here *byte* and the constant *Monday* has the value 0, Tuesday has the value 1, etc. The constant *Error* is initialized explicitly with the value 10.

In the following method, the user must enter a text: Mo, Tu, We, Th, Fr, Sa or Su. This text is then converted to an *enum* that is printed on the screen.



Click on the ad to read more

Download free eBooks at bookboon.com

```
static void Test1()
{
   Ugedag dag;
   Console.Write("(Mo, Tu, We, Th, Fr, Sa, Su)? ");
   string text = Console.ReadLine();
   switch (text)
   {
      case "Mo":
         dag = Ugedag.Monday; break;
      se "Tu":
         dag = Ugedag.Tuesday; break;
      case "We":
         dag = Ugedag.Wednesday; break;
      case "Th":
         dag = Ugedag.Thursday; break;
      case "Fr":
         dag = Ugedag.Friday; break;
      case "Sa":
         dag = Ugedag.Saturday; break;
      case "Su":
         dag = Ugedag.Sunday; break;
      default:
         dag = Ugedag.Error; break;
   }
   Console.WriteLine(dag);
}
```

Note the *switch* statement that switches on a string. It is allowed in C#. Note also that the program writes *Sunday*, that is the name of the constant.

An enum as above define 8 values, but that does not mean that a variable of that type takes up 8 bytes. It takes up only one byte, and it is only a question that there is assigned a name determined by the variable's value.

The default type is *int*, and you can simply write:

```
public enum Color
{
   Diamonds, Hearts, Spades, Clubs
}
```

Enum types are effective and it is a better solution than to define a number of constants for a concept, and enum types can be used to increase readability.

The above examples concerning enums are found in Exam36.

Download free eBooks at bookboon.com

19 Struct

The last value type is a *struct* that is a structure that resembles a class to the confusion. The main difference is that a *struct* is a value type. Below is a *struct* which represents a point in a coordinate system:

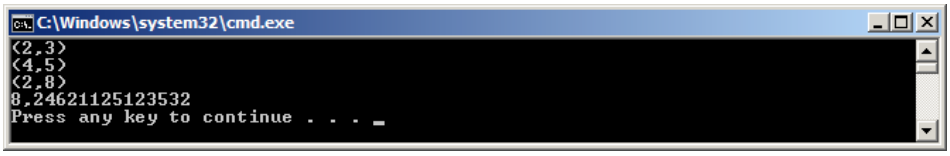
```
struct Point
{
   public double x;
   public double y;

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

   public double Length
   {
      get { return Math.Sqrt(x * x + y * y); }
   }

   public override string ToString()
   {
      return string.Format("({0},{1})", x, y);
   }
}
```

The type is called *Point*, and there are two instance variables of the type *double*. This means that every time you create a *Point* there is allocated 16 bytes on the stack. The type have a constructor, which initializes the two coordinates, and there is a single property, which returns the distance from the (0,0) to the point. In addition there is a *ToString()* method. Actually, there is also an implicit default constructor that sets both variables to 0 – this constructor can't be overridden. Below is a method that uses the type *Point*:

```
static void Test1()
{
   Point p1;
   Point p2 = new Point(4, 5);
   p1.x = 2;
   p1.y = 3;
   Point p3 = p1;
   p3.y = 8;
   Show(p1);
   Show(p2);
   Show(p3);
   Console.WriteLine(p3.Length);
}

private static void Show(Point p)
{
   Console.WriteLine("({0},{1})", p.x, p.y);
}
```


Download free eBooks at bookboon.com

If you run the program you get:



Explanation

First the method create a *Point*

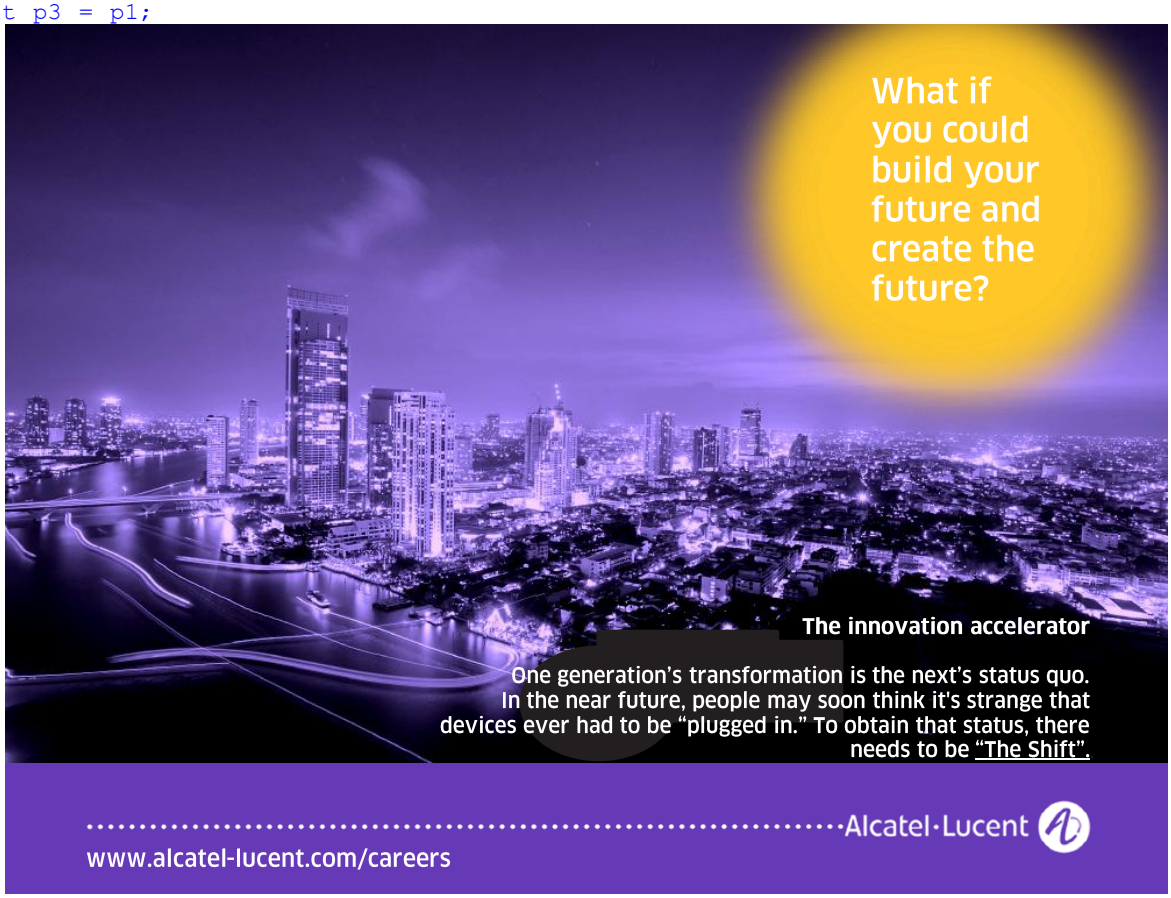
```
Point p1;
```

That means creating a variable *p1* on the stack and the default constructor is executed. Note that it is not a reference, but a variable on the stack takes up 16 bytes. Then the method create another variable *p2*, but this time with the *new* operator:

```
Point p2 = new Point(4, 5);
```

and even if *p2* is created with *new*, it is still created on the stack. The *new* operator is used here to get the constructor executed and therefore has a different meaning than it has with a class. As the next point values are assigned to the coordinates of *p1*. Note that it is possible, since both variables are *public*. Then the program create a third variable:

```
Point p3 = p1;
```



Download free eBooks at bookboon.com

This means that the program create another variable on the stack, which is a copy of *p1*. Note that *p3* is a new variable, which has the same value as *p1*, but this is not a reference to *p1*. It can be seen by changing the value of *p3*, and that this change does not affect *p1*.

Comment

The syntax for a struct which is as for a class other than that the word *class* is replaced by the word *struct*. A *struct* can have constructors just as a class, properties and methods, and the only difference here is that a struct always have a default constructor, which can't be overwritten. A struct can also implement an interface, but a struct can't inherit. Aside from that there is no difference.

The difference between a struct and a class is thus on the application. Typically, a struct is used to encapsulate a few simple data types to achieve a better performance equivalent to that it is a value type. Therefore you will often skip properties, making variables *public*, as is the case in the type *Point*. When an object is allocated on the stack, there can be only one reference to it, and the need for data encapsulation is not the same as for heap-allocated objects.

Copying struct's

When a struct type is a value type, it means that if you assign a variable to another variable, it is a really copy

```
Point p1 = new Point(2, 3);
Point p2 = p1;
```

where *p1* is copied to *p2*. You must be special aware of that if a struct has instance variables of reference type, since it is only the references there are copied and not the objects which they refer. If you are not aware of that, it can sometimes produce unexpected results. As an example, consider the class *Dice*:

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

   public Dice()
   {
      Throw();
   }

   public int Eyes
   {
      get { return eyes; }
   }

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


Download free eBooks at bookboon.com

There is nothing to note about it, besides it is a reference type. The following type defines a pair of die – a cup with two *Dice* objects:

```
struct Par
{
   public Dice d1;
   public Dice d2;

   public void Throw()
   {
      d1.Throw();
      d2.Throw();
   }

   public override string ToString()
   {
      return d1.Eyes + " " + d2.Eyes;
   }
}
```

It is a value type with two instance variables, both types of reference. This means that if you copy a *Par*, it's these references that will be copied:

```
static void Test2()
{
   Par p1;
   p1.d1 = new Dice();
   p1.d2 = new Dice();
   Console.WriteLine(p1);
   Par p2 = p1;
   Console.WriteLine(p2);
   p1.Throw();
   Console.WriteLine(p1);
   Console.WriteLine(p2);
}
```

The first line creates a *Par* object, and then the variables are initialized to new *Dice* objects. This is necessary because the default constructor does not create these objects and can't be overridden. In the fourth line another *Par* object is created, which is set equal to *p1*. Note that this means that the object *p1* is copied to *p2*, but what is copied, is the two instance variables, and they thus references to two *Dice* objects. The result is that the two *Par* objects, each has their own reference variables, but they refer to the same *Dice* objects. It becomes clear if you throw with one *Par* (*p1*):

Download free eBooks at bookboon.com

Nullable struct's

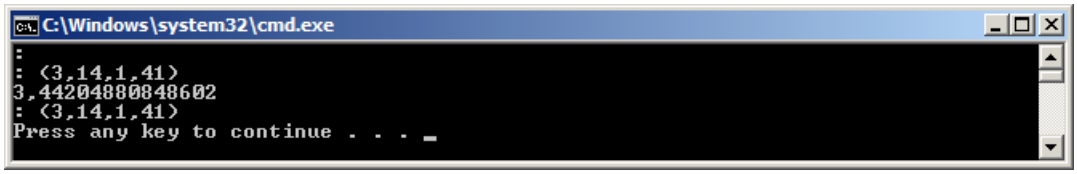
A *struct* is a value type, and therefore can't be *null*, but as shown above, a simple type may be *nullable*, and the same applies for a struct. If *Point* is the same type as above, consider the following:

```
static void Test3()
{
   Nullable<Point> p = null;
   Console.WriteLine(": " + p);
   p = new Point(3.14, 1.41);
   Console.WriteLine(": " + p);
   Console.WriteLine(p.Value.Length);
   Point v = p.Value;
   v.x = 11;
   v.y = 13;
   Console.WriteLine(": " + p);
}
```

Here, *p* is not a *Point* but a *Nullable<Point>*, a *Point*, which may be *null*. What happens is that the value type is encapsulated in a reference type. Above, *p* is initially *null*. Next, *p* is set to a new *Point* object, but you now have hidden the properties of thetype *Point*. Instead, we can refer to the encapsulated object with the property *Value*. Note especially that when you write

```
Point v = p.Value;
```

you get a copy of the encapsulated object.



One should therefore not make nullable variables, unless you have special needs. Note especially that

```
int? a
```

only is a short notation for

```
Nullable<int> a
```

The above examples concerning. struct's are found in Exam37.

Download free eBooks at bookboon.com

20 Generic types

Generic types are attached to the so-called collection classes that are dealt with later, but the goal here is to show how to write custom generic types. Short it can be said that the aim is to write types that are general and can be used in many contexts. Instead of the generic type one also refers to parameterized types corresponding to that it is types that depend on one or more parameters.

Generic methods

Before I show how to write a custom generic type, I will look at a related issue, namely what we mean by a generic method. As an example is shown a method to swap two integers of the type *int*:

```
static void Swap(ref int a, ref int b)
{
   int t = a;
   a = b;
   b = t;
}
```

It is a very simple method but a method of great use, for example if you have to sort an array. The method has a problem that is closely related to the type *int* in that way that if you also need a method that can swap two objects of the type *double*, then it is necessary to write a new *Swap()* method acting on *double* and similarly for all the other types in which there is a need for a *Swap()* method. It may therefore be desirable to write a general method that can handle all types. This is where generic methods come into play:

```
static void Swap<T>(ref T a, ref T b)
{
   T t = a;
   a = b;
   b = t;
}
```

It's also called a parameterized method similar to that of the method is associated with a type parameter, here called *T*. In addition to that there after the method name is a <*T*> it is equivalent to that *int* everywhere is replaced with type parameter *T*. For example the method can be used as follows, where it exchanges two strings:

```
string s1 = "Svend";
string s2 = "Knud";
Swap(ref s1, ref s2);
Console.WriteLine(s1);
Console.WriteLine(s2);
```


Download free eBooks at bookboon.com

As another example is below shown a generic method that prints an array:

```
static void Print<T>(T[] arr)
{
   foreach (T t in arr) Console.Write(t + " ");
   Console.WriteLine();
}
```

Note that the code is written entirely as *T* was an existing type – a concrete type. However, it has its limitations since the only thing you can do with objects of the type *T* is what you can with an *object* – the compiler can't impossible have knowledge of other properties of the type *T*. However, there are possibilities to impose restrictions on the type parameter *T*, so that the compiler can assume certain methods or properties.

Click on the ad to read more

Download free eBooks at bookboon.com

Exam38

Sorting an array

It should be written as a method which can sort an array of objects of any type.

How to

There are many different sorting methods, and here I will use a method that can be described as follows

- loop over the array and find the smallest element
- swap the smallest element with the element in position 0 – now the first item is correct
- loop over the last n-1 elements and find the smallest among these
- swap that element with the element in position 1 – now the first two elements are correct
- loop over the last n-2 elements and find the smallest among these
- swap that element with the element in position 2 – now the first three elements are in place
- continue now until the array is sorted – for each pass find the smallest of the elements that are not already in place and swap it to the right position

The result is that after k passes are the first k elements sorted while you still have to sort the last n-k elements. It is a very simple sorting method, but it is however not the most effective – at least not for large arrays.

Writing a method for sorting an array of any type, is a too large requirement, since a sorting of the elements will always include that the elements can be compared and ranked in order of size. Many of the built-in types can be, for example the simple types and the type *string* that can be compared with the comparison operators, but other types can also be compared, and it usually happens in that they implement an interface called *IComparable*. This interface defines only one method called *CompareTo()*, which has an *object* as a parameter. The protocol is that the method must return -1 if the current object is less than the parameter, 1 if the current object is greater than the parameter and otherwise 0. The interface is also available in a generic version and the method *CompareTo()* is thus also a generic parameterized with the kind of elements to be compared. The sum of all this is that the sorting method can be written as follows:

```
static void Sort<T>(T[] arr) where T : IComparable<T>
{
   for (int i = 0; i < arr.Length - 1; ++i)
   {
      int k = i;
      for (int j = i + 1; j < arr.Length; ++j)
         if (arr[j].CompareTo(arr[k]) < 0) k = j;
      if (i != k) Swap(ref arr[i], ref arr[k]);
   }
}
```


Download free eBooks at bookboon.com

Explanation

The method is simple and expresses the above algorithm, but there are a few important things to note. Note first that it is a generic method parameterized with *T*, and that it has a parameter *arr* that is an array of the type *T*. Next, note you should note the *where* part that expresses that the parameter type *T* must implement the interface *IComparable<T>* – thus a parameterized version of *IComparable*. Stated somewhat differently, the method can only work on arrays of types that implement this interface. If you try to apply the method to other types, you get a translation error. Note also how the method *CompareTo()* is used to compare elements, and note finally how the generic method *Swap()* is used.

Test

Below is an example of how the method can be used to sort an array with elements of the type *int*:

```
static void Test1()
{
   int[] t = { 17, 13, 29, 3, 19, 11, 5, 2, 7, 23 };
   Print(t);
   Sort(t);
   Print(t);
}

static void Print<T>(T[] arr)
{
   foreach (T t in arr) Console.Write("{0} ", t.ToString());
   Console.WriteLine();
}
```

Note that also the method *Print()* is generic.

When things go well, this is due to the type *int* which implements the interface *IComparable<int>*.

I will once again return to the type *Dice*, but this time with an extension so that it implements the *IComparable* interface:

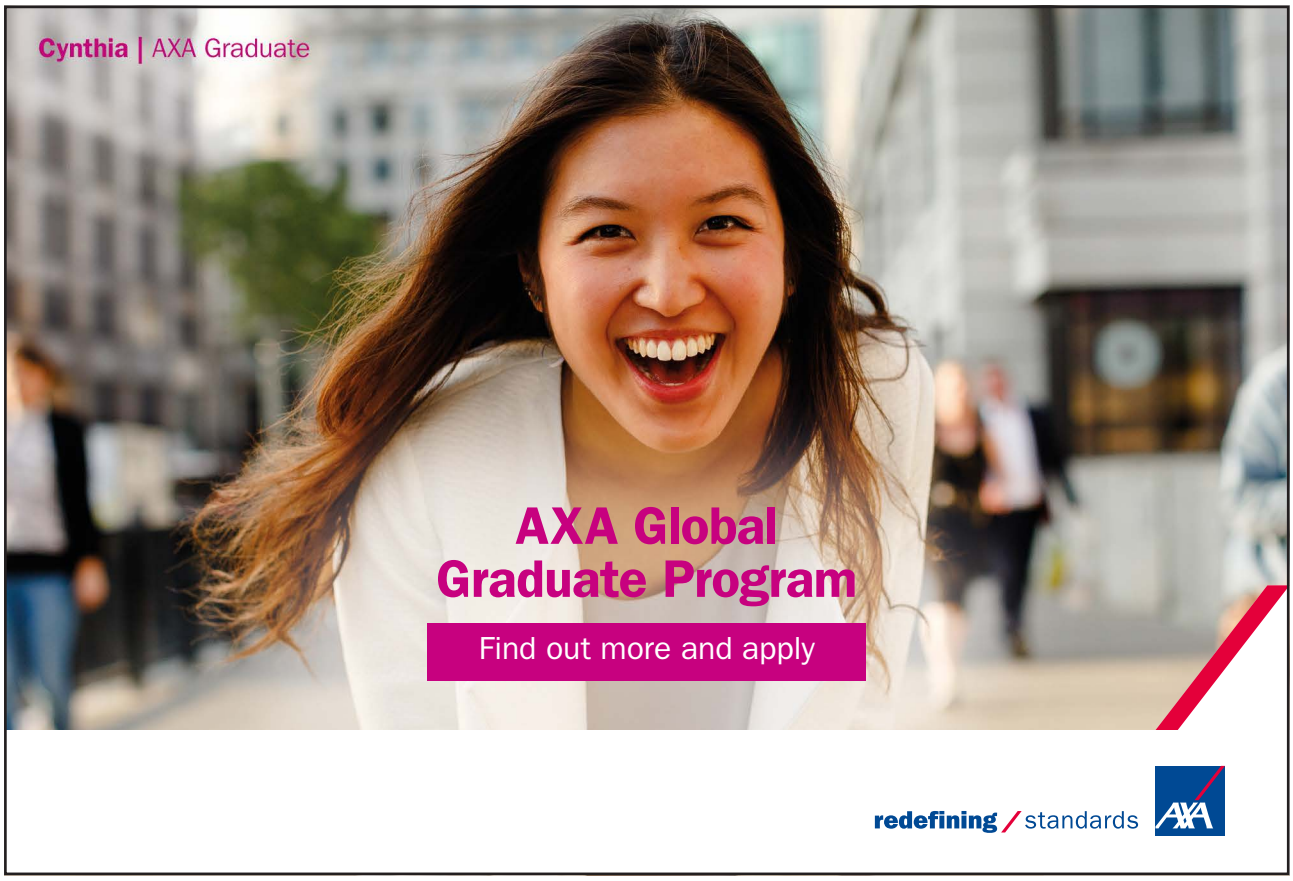
```
class Dice : IComparable<Dice>
{
   private static Random rand = new Random();
   private int eyes;

   public Dice()
   {
      Throw();
   }
```


Download free eBooks at bookboon.com

```
   public int Eyes
   {
      get { return eyes; }
   }

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

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

   public int CompareTo(Dice d)
   {
      return eyes < d.eyes ? -1 : eyes > d.eyes ? 1 : 0;
   }
}
```

This means that the cubes can now be arranged as such a two is less than a three, etc. Note that how they are ranked, is something that the programmer has specified in the implementation of the method *CompareTo()*, and that in principle one could have chosen any other arrangements.

As the next point I would like to create an array of cubes, but for this I will write a generic method that as a parameter has the size of the array:



Click on the ad to read more

Download free eBooks at bookboon.com

```
static T[] Create<T>(int n) where T : new()
{
   T[] arr = new T[n];
   for (int i = 0; i < arr.Length; ++i) arr[i] = new T();
   return arr;
}
```

Note first the use of *where*. It means that the parameter type *T* must have a default constructor – if it is not the case, one gets a translation error. The result is that when the array elements are created in the loop, you can be sure that they are properly initialized. Note that the class *Dice* satisfies that and has a default constructor.

Below is a code that creates an array with cubes, and sort it:

```
static void Test2()
{
   Dice[] b = Create<Dice>(10);
   Print(b);
   Sort(b);
   Print(b);
}
```

Here you should note that the method *Create()* is generic and that the parameter does not depend on the parameter type. If you just write the *Create(10)*, the translator can't know what *Create()* you wish to perform, and one must therefore set the parameter type after the method's name. In other examples, it is unnecessary (but legally) because the translator from the actual parameter can see what type it is. When, for example you write

```
Print(b);
```

the compiler can from the type of *b* to see what type the argument have, but it is legal to write

```
Print<Dice>(b);
```

Comment

In the examples above, I have shown two applications of the use of *where* to place restrictions on the type parameter. There are a few other cases:

- *where T : struct*
- *where T: class*

where the first means that the type parameter must be a value type, while the other means that the type should be a reference type. Finally I have in the method *Sort()* used that the type must implement an interface, but you can with the same syntax indicate that the type must inherit a class.

Download free eBooks at bookboon.com

Parameterized types

Also types can be generic, and just to show the syntax, I will start with a type that represents a pair of objects:

```
class Par<T1, T2>
{
   private T1 arg1;
   private T2 arg2;

   public Par()
   {
   }

   public Par(T1 t1, T2 t2)
   {
      arg1 = t1;
      arg2 = t2;
   }

   public T1 Arg1
   {
      get { return arg1; }
      set { arg1 = value; }
   }

   public T2 Arg2
      {
      get { return arg2; }
      set { arg2 = value; }
   }

   public override string ToString()
   {
      return string.Format("({0}, {1})", Arg1.ToString(), Arg2.ToString());
   }
}
```

It is an extremely simple type which is parameterized with the two type parameters. The class defines properties of the two variables and besides these properties the class have two constructors and a *ToString()*.

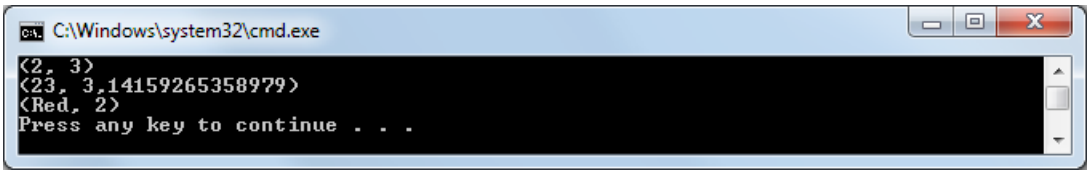
You should primarily note the syntax of a parameterize type, and that there may be one or more type parameters (this also applies to a parameterized method).

Download free eBooks at bookboon.com

Below is a program that uses the type:

```
class Program
{
   static void Main(string[] args)
   {
      Par<int, int> p1 = new Par<int, int>(2, 3);
      Par<int, double> p2 = new Par<int, double>();
      p2.Arg1 = 23;
      p2.Arg2 = Math.PI;
      Par<string, Dice> p3 = new Par<string, Dice>("Red", new Dice());
      Console.WriteLine(p1);
      Console.WriteLine(p2);
      Console.WriteLine(p3);
   }
}
```

There is not much to explain and if the method is executed, the result is the following:



The example is called Exam39.

Click on the ad to read more

Download free eBooks at bookboon.com

Exam40

The class Set

In this example I will show a type *Set* that implements the mathematical concept of a set. A set is a collection of objects which basically allows you to ask whether an object is in the set or not. Finally, the class should implements the basic set operations such as intersection, union and set difference. Exactly the type must have the following properties:

- that you can get to know how many elements the set contains
- that you can read the element in position *n*
- that you can add an element to the set
- that you can remove a particular element from the set
- that you can ask whether a certain element is in the set
- that you can form the union of the two sets
- that you can form the intersection of two sets
- that you can form the set difference of two sets

It should be a generic type, so that it is a set which can be applied to arbitrary objects.

How to

A set is defined as follows:

```
public interface ISet<T>
{
   int Count { get; }
   T this[int n] { get; }
   void Add(T elem);
   void Remove(T elem);
   bool Contains(T elem);
   ISet<T> Union(ISet<T> set);
   ISet<T> Intersection(ISet<T> set);
   ISet<T> Difference(ISet<T> set);
}
```

A set can then be defined as a class that implements this interface:

```
public class Set<T> : ISet<T>
{
   private T[] elems = new T[10];
   private int count = 0;

   public int Count
   {
      get { return count; }
   }
```


Download free eBooks at bookboon.com

```
public T this[int n]
{
   get { return elems[n]; }
}

public void Add(T elem)
{
   if (IndexOf(elem) >= 0) return;
   if (count == elems.Length) Expand();
   elems[count++] = elem;
}

public void Remove(T elem)
{
   int n = IndexOf(elem);
   if (n >= 0) elems[n] = elems[--count];
}

public bool Contains(T elem)
{
   return IndexOf(elem) >= 0;
}

public ISet<T> Union(ISet<T> set)
{
   Set<T> tmp = new Set<T>();
   for (int i = 0; i < set.Count; ++i) tmp.Add(set[i]);
   for (int i = 0; i < count; ++i) if (!set.Contains(elems[i])) tmp.Add(elems[i]);
   return tmp;
}

public ISet<T> Intersection(ISet<T> set)
{
   Set<T> tmp = new Set<T>();
   for (int i = 0; i < count; ++i) if (set.Contains(elems[i])) tmp.Add(elems[i]);
   return tmp;
}

public ISet<T> Difference(ISet<T> set)
{
   Set<T> tmp = new Set<T>();
   for (int i = 0; i < count; ++i) if (!set.Contains(elems[i])) tmp.Add(elems[i]);
   return tmp;
}

public override string ToString()
{
   StringBuilder builder = new StringBuilder();
   builder.Append("{");
   for (int i = 0; i < count; ++i)
   {
      builder.Append(' ');
      builder.Append(elems[i]);
   }
   builder.Append(" }");
   return builder.ToString();
}

private int IndexOf(T elem)
{
   for (int i = 0; i < count; ++i) if (elems[i].Equals(elem)) return i;
   return -1;
}
```


Download free eBooks at bookboon.com

```
   private void Expand()
   {
      T[] temp = new T[2 * elems.Length];
      for (int i = 0; i < count; ++i) temp[i] = elems[i];
      elems = temp;
   }
}
```

Explanation

Note, first, that the interface is generic, and hence that an interface in the same manner as a class can be defined generic. There is not much to explain about it, apart from that one anywhere use the parameter type *T* as if it were a concrete type.

The implementation has only a default constructor, which creates an empty set. In addition, the *Add()* method tests that the same item is not added twice, and if the item is already there, nothing happens. The same applies to *Remove()*, that if you try to delete an element not found in the set, the operation is just ignored. One can discuss these choices and you could instead have chosen a solution where the user is in one way or another (for example as a return value) could be notified if the operation was not performed correctly. The goal with my implementation is mainly to make the code simple and easy to read, but it is not very efficient algorithms.

The type is generic, parameterized by the type parameter *T*, and a set can then contain elements of the type *T*. There are no requirements of the type parameter – there is no *where* part. That is that the type *Set<T>* can be used for all types of objects. Note, however, the method *IndexOf()*, which is a private method that finds the location of a particular element in the array. It uses the method *Equals()* to compare elements. That means that if everything should works as intended, the parameter type *T* must implement the method *Equals()* with value semantic.

Most of the code is directly out of the road, but there is one thing that you should note. The type is implemented by means of an array, as the container that contains the elements of the set. When a set is created, is allocated space for 10 elements, which seem rather arbitrary, but this is also what it is and the question is what should happen if you try to add more than 10 elements to the set. Here a used a principle where the capacity is doubled if you try to add elements beyond the current capacity. It consists in create an array of double size, and copy the old array to the new. It is handled by the method *Expand()*, which possibly is called from the *Add()* method. It is a simple implementation which means that a set in principle has no upper limit on the number of elements – and which is also used by the collection classes.

Download free eBooks at bookboon.com

Comment

If you look at the methods for union, intersection and set difference they are not very effective. There are simply too many loops inside each other. It is perhaps not entirely obvious, but if you examine more closely what is happening, the problem comes to light. The problem is in fact the method *IndexOf()*, which consisting of a loop which passes through all the elements of the set. Consider as an example the following statement (from the method *Union()*):

```
for (int i = 0; i < count; ++i) if (!set.Contains(elems[i])) tmp.Add(elems[i]);
```

It consists of a loop, which runs through the sets elements. In principle it is ok, but for each element the statement called *Contains()* which calls *IndexOf()*, which then performs a second loop, and it gives a bad performance. It is said that the algorithm has a poor time complexity.

It is possible to do it better if organizing the elements more clever than just adding them to an array, but it overshoots the target for this book and is not the theme for this example. The goal here is to give an example of a generic type with some utility, and contains the set not too many elements, it works very well indeed.

Click on the ad to read more

Download free eBooks at bookboon.com

Comment

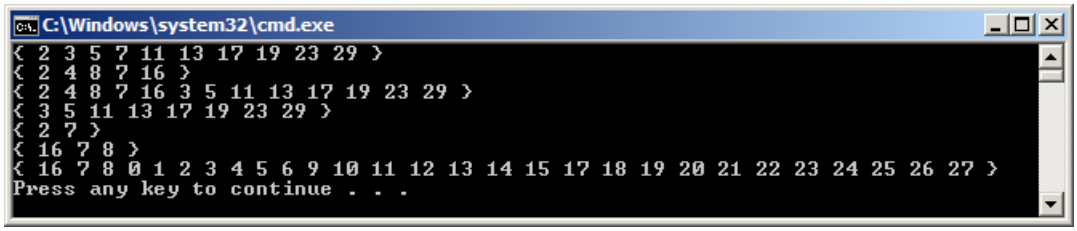
As explained above, the class *Set<T>* is implemented so that its methods are not quite as effective as we would wish. That's exactly why you should program to an interface. If you respect it, and if one in its program everywhere know and use the type by the defining interface, so you can later implement the *Set* class in a different and more efficient manner, without affecting the applications that use a *Set*.

Test

Below is a method used to test the class:

```
static void Test1()
{
   ISet<int> A = new Set<int>();
   A.Add(2);
   A.Add(3);
   A.Add(5);
   A.Add(7);
   A.Add(11);
   A.Add(13);
   A.Add(17);
   A.Add(19);
   A.Add(23);
   A.Add(29);
   ISet<int> B = new Set<int>();
   B.Add(2);
   B.Add(4);
   B.Add(8);
   B.Add(7);
   B.Add(16);
   Console.WriteLine(A);
   Console.WriteLine(B);
   ISet<int> C = A.Union(B);
   ISet<int> D = A.Difference(B);
   ISet<int> E = A.Intersection(B);
   Console.WriteLine(C);
   Console.WriteLine(D);
   Console.WriteLine(E);
   B.Remove(2);
   B.Remove(4);
   B.Remove(6);
   Console.WriteLine(B);
   for (int i = 0; i < 28; ++i) B.Add(i);
   Console.WriteLine(B);
}
```


Download free eBooks at bookboon.com



Also a struct may be generic. Below is a type that implements a generic point, but in which the coordinates must be a value type:

```
public struct Point<T> where T : struct
{
   public T x;
   public T y;

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

   public override string ToString()
   {
      return string.Format("({0}, {1})", x, y);
   }

   public override bool Equals(object obj)
   {
      if (!(obj is Point<T>)) return false;
      Point<T> p = (Point<T>)obj;
      return p.x.Equals(x) && p.y.Equals(y);
   }
}
```

Note also that the *Equals()* method is overloaded, so the type can be applied to elements of a Set.

Below is a simple application of the class:

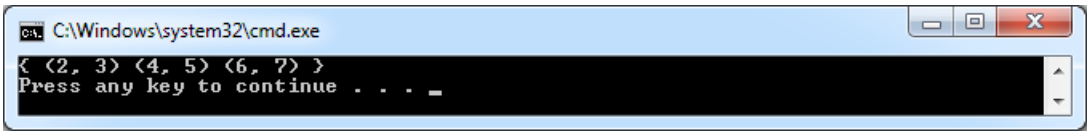
```
static void Test2()
{
   Point<int> p1 = new Point<int>(2, 3);
   Point<double> p2 = new Point<double>(1.41, 3.14);
   Point<Point<int>> p3 =
   new Point<Point<int>>(new Point<int>(1, 4), new Point<int>(2, 5));
   Console.WriteLine(p1);
   Console.WriteLine(p2);
   Console.WriteLine(p3);
}
```

Here you must particularly note the point *p3*, whose coordinates are of type *Point<int>*, which is legal, but perhaps hardly makes any sense.

Download free eBooks at bookboon.com

As a final example below shows a method that creates a *Set* of points:

```
static void Test3()
{
   ISet<Point<int>> A = new Set<Point<int>>();
   A.Add(new Point<int>(2, 3));
   A.Add(new Point<int>(4, 5));
   A.Add(new Point<int>(6, 7));
   Console.WriteLine(A);
}
```



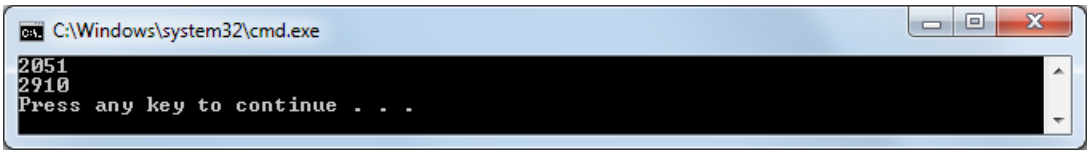
Efficiency

I mentioned above, that the class is not very efficient. To examine the effectiveness, I would try the following method:

Click on the ad to read more
Download free eBooks at bookboon.com

```
static void Test4()
{
   int N = 10000;
   ISet<int> A = Create(N, 2 * N);
   ISet<int> B = Create(N, 2 * N);
   Stopwatch sw = new Stopwatch();
   sw.Start();
   ISet<int> C = A.Union(B);
   sw.Stop();
   Console.WriteLine(sw.ElapsedMilliseconds);
   sw.Start();
   ISet<int> D = A.Intersection(B);
   sw.Stop();
   Console.WriteLine(sw.ElapsedMilliseconds);
}

static ISet<int> Create(int n, int m)
{
   Set<int> S = new Set<int>();
   while (n-- > 0) S.Add(rand.Next(m));
   return S;
}
```

Here is *Create()* a method that creates a set of *n* elements of the type *int*, which lies between 0 and *2n*. The method *Test4()* uses this method to create two sets each with 10000 elements. Then it use the class *Stopwatch* (defined in the namespace *System.Diagnostics*) to measure how long (in milliseconds) it takes to form the union, and finally repeats the method on intersection. If the method is executed, the result could be as follows:



This means that each operation takes between 2 to 3 seconds (on my machine).

Download free eBooks at bookboon.com

21 Exception handling

When writing a program, there may be errors. For example you can write the program code incorrectly, so the program can't be translated. This is because the program is written with an incorrect syntax, and that kind of mistake is rarely unproblematic, as they are caught by the compiler. One speaks therefore also of a compiler error. These errors must obviously be addressed, which – until you get trained – can be difficult enough, but when I call them unproblematic, it is because the compiler can find them, and the program can't execute before they are corrected.

Another type of errors are logical errors where the program can be translated and run, but when it does something else than the idea was. It may, for example be a calculation that gives a wrong result or an incorrect value that is stored in a database. It's the hardest errors because they can't be caught by the compiler, and because the program can actually run for a period before the error is acknowledged, but also because it may be errors that are difficult to locate. The user of the program detects the symptom, but it can be hard to find where in the code it is that it goes wrong. The remedy for this kind of errors is test, test and test again. To test a program is by no means simple, and it requires both time and procedures.

A third kind of error is caused by environmental conditions and are errors you as a programmer can't really guard against, but conversely also errors that you in one way or another must take care of. As an example, it may be a user who enters something – for example a number – and you have no control over what the user enters. As another example, one can imagine a program that will use a file that does not exist. In those situations, the program will handle the error, which means that it must be able to find that there is an error and if so, decide what is to happen. Typically it will be such that a method can capture that there is an error, but the method can't know how the error is handled, but it must instead let the place from where it was called, know that there has been an error and leave the error handling to the calling code. One way to solve the problem is to let the method that can detect the error return an error code and the calling code can then test the error code and take an appropriate action determined by the error code. This strategy is really good, but it can only be used in situations where the method would not else have a return value.

Another strategy is to use *exception handling*. The idea is pretty simple. A method can, in the case of a fault throw an exception, and if it does so, the method is immediately interrupted. The place from where the method was called can then choose to catch the exception and take an appropriate action. Consider as an example (labeled Exam41) the following method that calculates the ratio of two integers:

```
static int Div(int a, int b)
{
   if (b == 0) throw new ApplicationException("Division med 0...");
   return a / b;
}
```


Download free eBooks at bookboon.com

One can, as we know not divide by 0. If *b* is 0, the method can't perform the calculation, but it may also not know what to do. It can't just return something, because it could by the calling code be interpreted as a result that it was illegal. The method may test the value of *b*, and if it is 0 it raise an exception. This means that the method stops with an exception that is sent to the calling code. Below is a code that uses the above:

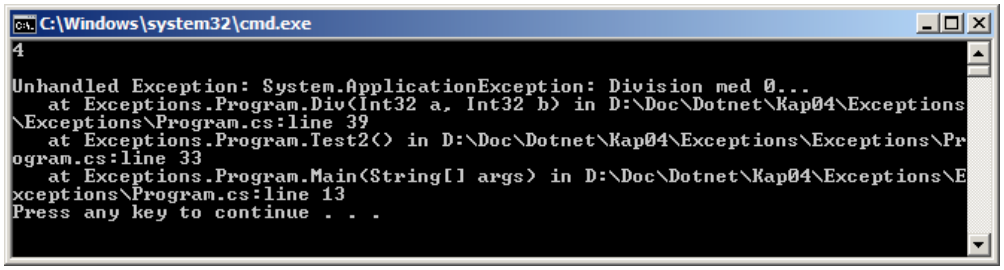
```
static void Test1()
{
   try
   {
      Console.WriteLine(Div(23, 5));
      Console.WriteLine(Div(23, 0));
   }
   catch (ApplicationException ex)
   {
      Console.WriteLine(ex.Message);
      Console.WriteLine("The method is completed with an error...");
   }
}
```

The code that may raise an exception – here the method *Div()* – is placed in a try block. If *Div()* raises an exception, the control is transferred to the subsequent *catch* handler that then performs en error handling, which here is merely to print a message on the screen.

An exception is a type, which is a class. Above is the type *ApplicationException*, but there are other options. For an exception can be caught in the calling code, there must be consistency between the type of the exception that is raised – the type after the *throw* – and type after *catch*.

The calling code does not need to catch an exception. Is it the case and an exception are not treated at any level, the program will be terminated with a default exception handling:

```
static void Test2()
{
   Console.WriteLine(Div(23, 5));
   Console.WriteLine(Div(23, 0));
}
```



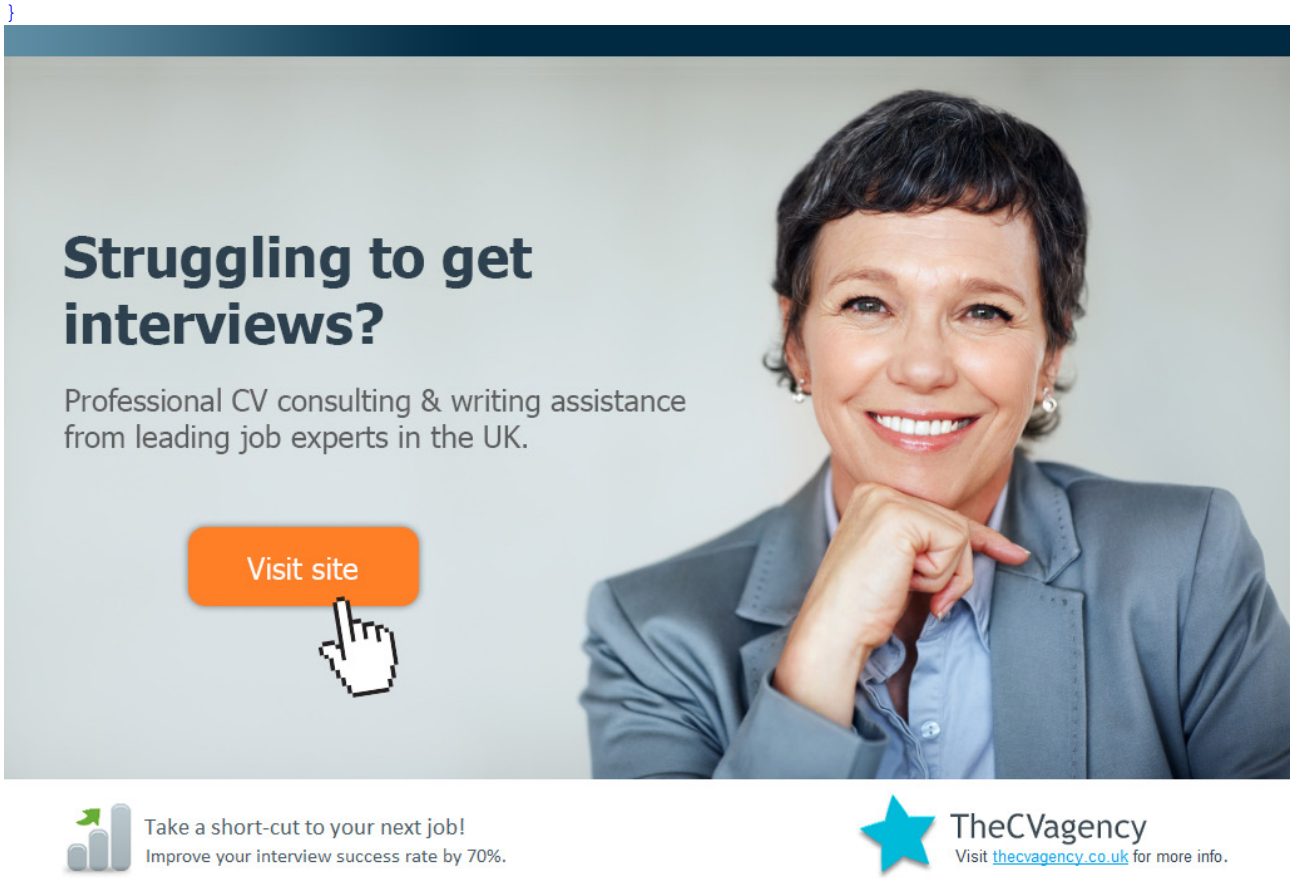
Download free eBooks at bookboon.com

As mentioned, there are multiple exception types, actually many and most of them are classes in the .NET Framework that may raise a variety of exceptions. The fundamental base class is called *Exception*, and belongs to the *System* namespace. *ApplicationException* is a derived class, and although a custom code may as well raise an *Exception* (and often do), it is the thought that the type of an exception raised by a custom method must either be an *ApplicationException* or a type derived there from.

As an example of how the exception handling works in C#, I will use the generic class, defined above. The class has a method *Add()*, and a method *Remove()*, both of which is inappropriate. The problem is that for both methods it is possible that they can't perform the requested operation (an element can't be added to the set, if it is already there, and an element can't be removed if it is not in the set), and if it is the case nothing happens. It is an unfortunate solution, since the user does not receive a notification if the operation is not performed. The main problem is that the two methods may test whether the operation can be performed, but they can't know what to do, if the operation could not be done. The solution is to let the methods raise an exception if the operation can't be performed.

I will start with the following exception type:

```
public class SetException : ApplicationException
{
   public SetException(string message)
      : base(message)
   {
   }
}
```

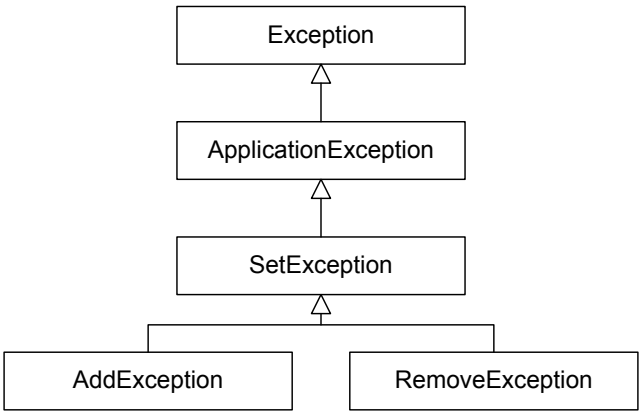


Download free eBooks at bookboon.com

Here you should note that the type inherits *ApplicationException* and the only thing that happens is that there is a constructor that sends a text on to the base class's constructor. It is a very simple type, which in reality does not extend *ApplicationException* and the only purpose is to get a type that is specific for the current problem. I will define two additional types of exceptions, both types that inherit *SetException*:

```
public class AddException : SetException
{
   public AddException()
      : base("The element already exists in the set and can not be added")
   {
   }
}

public class RemoveException : SetException
{
   public RemoveException()
      : base("The element is not found in the set and can not be removed")
   {
   }
}
```

From the names and the text it is clear what they should be used for and the important thing is that there is the following class hierarchy are available:



I will now extend the code in the class *Set*, so it possibly raises an exception, but I have only shown the methods that have been changed:

```
public class Set<T> : ISet<T>
{
   private T[] elems = new T[10];
   private int count = 0;

   public void Add(T elem)
   {
      if (IndexOf(elem) >= 0) throw new AddException();
      if (count == elems.Length) Expand();
      elems[count++] = elem;
   }
```


Download free eBooks at bookboon.com

```
   public void Remove(T elem)
   {
      int n = IndexOf(elem);
      if (n < 0) throw new RemoveException();
      elems[n] = elems[--count];
   }
      .....
}
```

The methods *Add()* and *Remove()* now raise an exception if the operation can't be performed. Note that the exceptions there are raised, is of a different type.

Below is a method that uses the class *Set* and catch any exceptions:

```
static void Test3()
{
   ISet<int> A = new Set<int>();
   try
   {
      for (int i = 0; i < 20; ++i)
      {
         try
         {
            int t = rand.Next(30);
            A.Add(t);
            t = rand.Next(30);
            A.Remove(t);
         }
         catch (AddException ex)
         {
            Console.WriteLine(ex.Message);
         }
         catch (RemoveException ex)
         {
            Console.WriteLine(ex.Message);
         }
         catch (SetException ex)
         {
            Console.WriteLine(ex.Message);
         }
         finally
         {
            Console.WriteLine(A);
         }
      }
      Print(A, A.Count);
   }
   catch
   {
      Console.WriteLine("There was an error");
   }
}
```


Download free eBooks at bookboon.com

Here are some things to be explained. First, notice that there are two *try* blocks: An outer outside the *for* loop, and an inner encapsulating the body of the *for* loop. So please note that you can have *try* blocks inside each other. In the loop, two things happens:

1. adding a random number (between 0 and 29) the set A
2. a random number is removed from the set

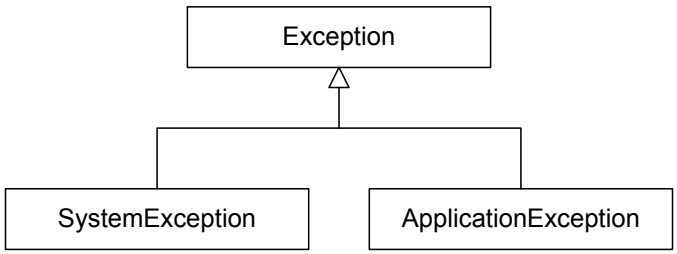
In both cases there may be an exception. Therefore, the code is placed in a try block. After the *try* block, there are three *catch* handlers, which are intended to illustrate that there may be several *catch* handlers for a *try* block. If the block raises an exception, the program control is transferred to the *catch* handler that matches the type of the exception raised. It is, therefore, that exceptions may have a different type, so that multiple *catch* handlers can control what happens in different situations.

Also note that the innermost *try* block has an associated *finally* handler. This will – if it is there what is not necessary – be performed regardless of whether there is an exception or not.

Finally, there is the outer try block. This is an example of an anonymous *catch* handler, and thus a *catch* handler that is executed when there is an exception and regardless of type – it *catch* anything.

SystemException

When an application raises an exception, it should always be of type *ApplicationException* or a type that is directly or indirectly derived from *ApplicationException*. The class is, as mentioned derived from the class *Exception* and there is nothing that prevents that one can raise an exception of the type *Exception*. When you should not do it, it is because there is another exception class called *SystemException* that are also derived from *Exception*. Exceptions of this type are intended for exceptions raised by the runtime system or the basic classes from the framework, and they indicate fundamental and serious problems where the typical handling is to terminate the program with an error message. The following hierarchy separates the exceptions into two categories, one category indicate failure, which typically results in that the program exits with an error message while the second category indicates the error caused by the program's environment or use, and thus an error that should be handled by the program.

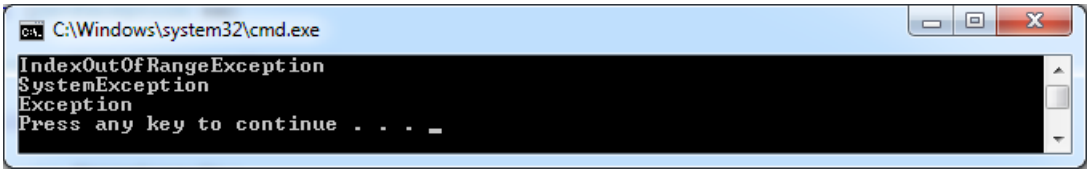


Download free eBooks at bookboon.com

As an example is below shown a method which results in an error, because it are indexing outside an array:

```
static void Test4()
{
   int[] t = { 2, 3, 5, 7, 11, 13, 17, 19 };
   try
   {
      int s = 0;
      for (int i = 0; i < 10; ++i) s += t[i];
      Console.WriteLine(s);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex.GetType().Name);
      Console.WriteLine(ex.GetType().BaseType.Name);
      Console.WriteLine(ex.GetType().BaseType.BaseType.Name);
   }
}
```

The result is as follows:




Click on the ad to read more

Download free eBooks at bookboon.com

22 Comments

Consider as an example the following class, which you have also seen in Exam34:

```
using System;
using System.Text;

/*
   This namespace contains several classes with miscellaneous methods.
   One can perceive the namespace as a custom class library.
*/
namespace Exam42
{
   /// <summary>
   /// Class with static methods for text operations.
   /// </summary>
   public static class String
   {
      /// <summary>
      /// Method, which cuts off a string to a maximum width.
      /// If the width does not exceed the desired width,
      /// the method simply returns the string unchanged.
      /// </summary>
      /// <param name="text">The string that must be cut</param>
      /// <param name="length">The maximum width of the return string</param>
      /// <returns>A string whose width is less than or equal to the length</returns>
      public static string Cut(string text, int length)
      {
         if (text.Length > length) return text.Substring(0, length);
         return text;
      }

      /// <summary>
      /// Method to adjust a text string within a given field with the width length.
      /// If the length of the text is larger than length, the method simply
      /// returns the text.
      /// Otherwise the method returns a string of length length, where text
      /// is left-aligned, and where the field is filled with the character fill.
      /// </summary>
      /// <param name="text">The text to be adjusted</param>
      /// <param name="length">Field width</param>
      /// <param name="fill">Fill character</param>
      /// <returns>Justifyed text</returns>
      public static string FillRight(string text, int length, char fill)
      {
         if (text.Length >= length) return text;
         StringBuilder builder = new StringBuilder(text, length);
         while (builder.Length < length) builder.Append(fill);
         return builder.ToString();
      }

      /// <summary>
      /// Method to adjust a text string within a given field with the width length.
      /// If the length of the text is larger than length, the method simply
      /// returns the text.
      /// Otherwise the method returns a string of length length, where text
      /// is right-aligned, and where the field is filled with the character fill.
      /// </summary>
```


Download free eBooks at bookboon.com

```
      /// <param name="text">The text to be adjusted</param>
      /// <param name="length">Field width</param>
      /// <param name="fill">Fill character</param>
      /// <returns>Justifyed text</returns>
      public static string FillLeft(string text, int length, char fill)
      {
         if (text.Length >= length) return text;
         // here are created a StringBuilder with the required capacity to the fill
         // characters you could instead have used the method Insert(), but it has a
         // worse complexity than Add()
         length -= text.Length;
         StringBuilder builder = new StringBuilder(length);
         while (builder.Length < length) builder.Append(fill);
         return builder.ToString() + text;
      }

      /// <summary>
      /// Method to adjust a text string within a given field with the width length.
      /// If the length of the text is larger than length, the method simply
      /// returns the text.
      /// Otherwise the method returns a string of length length, where text
      /// is center-aligned, and where the field is filled with the character fill.
      /// </summary>
      /// <param name="text">The text to be adjusted</param>
      /// <param name="length">Field width</param>
      /// <param name="fill">Fill character</param>
      /// <returns>Justifyed text</returns>
      public static string FillCenter(string text, int length, char fill)
      {
         if (text.Length >= length) return text;
         return
            FillRight(FillLeft(text, (text.Length + length) / 2, fill), length, fill);
      }
   }
}
```

I have this time inserted comments in the code. Comments have no effect on the translated program, but will only have affect for us humans to read and understand the code. Much can be said about comments, so some words about it.

It is important with comments in the code, and more than that – it should actually be a permanent part of the programming task quite on pair with writing the code itself, and there are at least two reasons:

1. All program code must be maintained over time, and often by other than the one who wrote the code. Therefore it is extremely important that the code has comments that tell about important decisions, and why the code is written as it is. Comments are important not only for others but also because you do not remember your own code even if it is just a few months old.
2. The actual process of commenting the code is important because during this process, you think through the code and wonder why the code is written as it is. It is an extremely efficient method to find errors and discrepancies in the code – and get them corrected.

Download free eBooks at bookboon.com

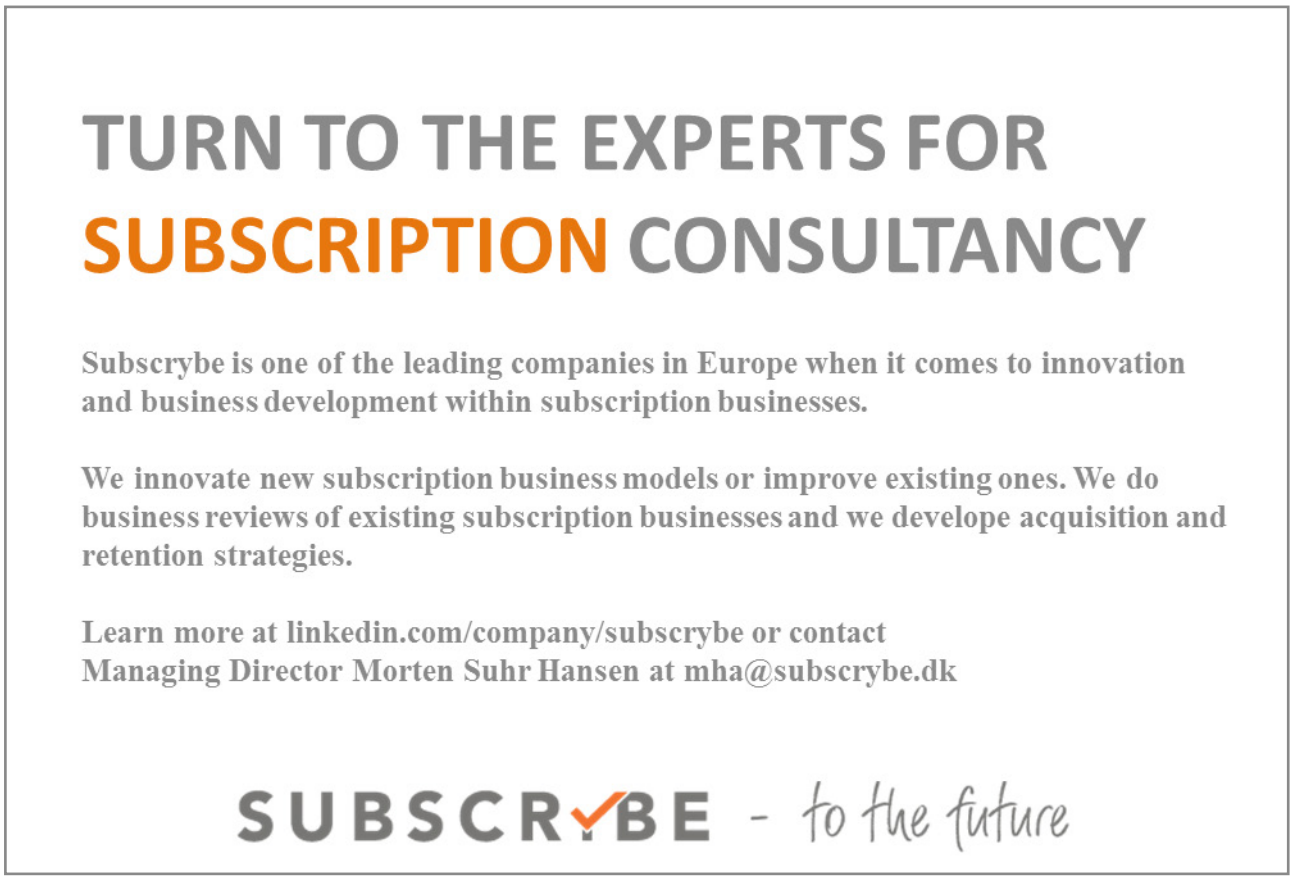
If you see my examples, you will in most cases not see any comments. This is partly because they are small examples and not actual applications that must solve practical everyday problems, and secondly, that the code is precisely explained in this book. It is true to say that a part of the books content could instead be comments in the examples.

In C# you have three kinds of comments. Above I have written a comment before the namespace:

```
/*
   This namespace contains several classes with miscellaneous methods.
   One can perceive the namespace as a custom class library.
*/
```

It's a little older kind of comment that has been inherited from the C programming language, but the characters /* start a comment and it will continue until you meet the characters */ and between these two markers can be all the text that you may have like on a single line or spread over several lines. This kind of comment is not used as often anymore, but there's nothing wrong with it, and if you need at the beginning of a source file to write a lengthy documentation, it is an excellent form of commentary.

The most common type of comment in C# is used in front of each method. This is partly due that Visual Studio auto generate a skeleton, which you must complete. If, for example you place the cursor in front of the method *FillCenter()* and press the / three times, Visual Studio generates the following skeleton to a comment:



Click on the ad to read more

Download free eBooks at bookboon.com

```
/// <summary>
///
/// </summary>
/// <param name="text"></param>
/// <param name="length"></param>
/// <param name="fill"></param>
/// <returns></returns>
```

Here, the programmer has to comment the following:

- A description of the method and what it does and including generally what the user of the method needs to know.
- An explanation of the method's parameters, including which rules (pre-conditions) that the parameters must satisfy.
- An explanation of the method's return value.

The advantage of using this kind of documentation – not just to methods, but for all program elements – is

- that the documentation has a standard form and that you remember to document all important elements as parameters and return value
- that the documentation is used by IntelliSense in Visual Studio
- that the documentation is in XML form, and therefore can use of a tool to form a complete documentation for an entire program, for example as HTML

The last type of comment is simpler and consists of everything after // and to the end of the line is a comment (see the method *FillLeft()* above). This comment is typically used to document the individual statements in for example a method, or the description of a variable or its equivalent.

It is easy to write comments, but quite another thing is what you should write, and there are many attitudes, and the following must then be mine. Generally, you should write what you believe that others and including even you self needs to know to read and understand the code and thus could maintain it. You should not write what is clear. You must assume that who must read the code knows the language, and you should not document the language itself, but the explanations for the choices made – you must explain how the algorithms are used and how they work. Are there solutions that are difficult to comprehend, then you should add explanatory comments. It is also wise to always document the variables and what they used for – at least instance variables. It can also be a good idea to add a comment, telling about modification of the code, when changes are made, to whom and why, and of course what's changed.

Download free eBooks at bookboon.com

You should special be consistent about the auto-generated comments and include them – at least for all *public* program elements. It is especially important for class libraries, which often must be used by anyone other than the programmer who wrote the classes. It can be hard to write that kind of evidence simply because it can be hard to find something to write (many methods and properties are obvious and self explanatory) and you often think that you do not have anything to write. Yet it is a place where you should be consistent and include these comments. One should be aware that this kind of comments is intended for those who must use the code, and not to those who need to maintain the code.

The program's code readability is extremely important, and you can even go so far that the code that is not easy to read is worthless. However, you can do many things to make the code readable than writing comments and including the following few guidelines:

- a block starts on a new line containing only the character {
- from the next line makes an indentation of two characters
- when a block ends repealed the indentation, and you move two characters to the left
- a block always end on a new line containing only the character }
- add always a space on either side of an operator
- a variable name always starts with a lowercase letter
- a name of a method, property, and a user-defined type always starts with a capital letter

Click on the ad to read more
Download free eBooks at bookboon.com

- be consistent in capitalization
- use good and explanatory names, but not for long names – they are hard to read
- use blank lines where you think it increases the readability

And so be consistent and have a style. Guidelines are good, but there will always be places where you may depart from them, but if you do it consistently, it is excellent. The above guidelines are to make the code self-documenting, and one can say that the comments should be used if the code can't explain itself.

One can hardly say enough about the importance of writing readable program code, but in terms of comments, you can also go too far. Generally I feel that a comment inside a method makes the code harder to read. It can be difficult to see what is program code and what's comments – the comments shadows the code. Comments inside the code I include only where I think they are absolutely necessary – and it is certainly often the case. The conclusion is that documentation is important, but exaggeration may have the opposite effect.

Download free eBooks at bookboon.com

23 Extension methods

If you want to extend a class with new methods, the approach is to write a derived class that adds the new methods. It is still the "right" strategy, but it is not always possible, for example if the class is *sealed* – it is a class that you can't inherit. One can however achieve the same thing with an extension method.

Consider the following class that defines three static methods to integers:

```
public static class Integer
{
   public static long DiffSum(this int n)
   {
      return (n + 1L) * n / 2;
   }

   public static int Add(this int n, params int[] t)
   {
      int s = n;
      for (int i = 0; i < t.Length; ++i) s += t[i];
      return s;
   }

   public static bool IsPrim(this int n)
   {
      if (n == 2 || n == 3 || n == 5 || n == 7) return true;
      if (n < 11 || n % 2 == 0) return false;
      for (int k = 3, m = (int)Math.Sqrt(n) + 1; k <= m; k += 2)
         if (n % k == 0) return false;
      return true;
   }
}
```

The first determines the sum of the numbers 1 + 2 + 3 + 4 + … + N. This can be done with a loop, but you can also use a formula as has been done above. The second method returns the sum of a series of integers, while the latter method tests whether an integer is a prime. Since all the methods are static, they may be carried out as follows:

```
Console.WriteLine(Integer.Add(2, 3, 5, 7, 11, 13, 17, 19));
Console.WriteLine(Integer.DiffSum(100));
Console.WriteLine(Integer.IsPrim(97));
```

which is not strange. You should however note that the class is static, and that the first parameter to each of the three methods is of the type *int*, and the declarations of these parameters are prefixed with the word *this*. It is the two factors that make that make the methods to extension methods. This means that methods can be performed as if they were instance methods defined for type *int*:

Download free eBooks at bookboon.com

```
int a = 2;
Console.WriteLine(a.Add(3, 5, 7, 11, 13, 17, 19));
int b = 100;
Console.WriteLine(b.DiffSum());
int c = 97;
Console.WriteLine(c.IsPrim());
```

and not only that – the methods are known to Intellisense in Visual Studio.

Apparently the type *int* is extended with new methods, but it is obviously not the case. An extension method is a usual static method, and it should be written in the same way as other static methods and can't refer for instance members of the class to which it is an extension. There are only talking about that with the word *this* in front of the first parameter it allows to use a method with same syntax as if it were an instance method. If you compare the above applications of the methods in the class *Integer*, it is clear that it is only a question of how to specify the first parameter – as a normal value or by using dot notation.

Extension methods have their uses, and is as such used by Microsoft a great in relation to LINQ.

In the previous section I showed a class *String*. It was a static class with static methods, where the first parameter in all methods has the type *string* (that is the type *System.String*). It is therefore extremely simple to modify these methods to extension methods for the *String* class – it's just adding the word *this* in front of the first parameter to all methods:


Click on the ad to read more

Download free eBooks at bookboon.com

```
public static class Str
{
   public static string Cut(this string text, int length)
   {
      if (text.Length > length) return text.Substring(0, length);
      return text;
   }

   public static string FillRight(this string text, int length, char fill)
   {
      if (text.Length >= length) return text;
      StringBuilder builder = new StringBuilder(text, length);
      while (builder.Length < length) builder.Append(fill);
      return builder.ToString();
   }

   public static string FillLeft(this string text, int length, char fill)
   {
      if (text.Length >= length) return text;
      length -= text.Length;
      StringBuilder builder = new StringBuilder(length);
      while (builder.Length < length) builder.Append(fill);
      return builder.ToString() + text;
   }

   public static string FillCenter(this string text, int length, char fill)
   {
      if (text.Length >= length) return text;
      return
         FillRight(FillLeft(text, (text.Length + length) / 2, fill), length, fill);
   }
}
```


Download free eBooks at bookboon.com

Part 3 Collection classes

All modern languages, including C# has a selection of collection classes. A collection class can be thought of as a container that can contain objects. The individual collection classes differ in how they organize objects, and what you can do with them, ie which access there is to the content of the container. The following describes the most important collection classes that are contained within the .NET framework.

.NET is born with commonly used collection classes fully implemented and ready to use. They are all implemented as generic classes, and I will in this part of the book mention the following:

- *List<T>*
- *Stack<T>*
- *Queue<T>*
- *LinkedList<T>*
- *Dictionary<K, V>*
- *SortedDictionary<K, V>*

I will only focus on how the classes can be used, and for a full documentation of the classes and their methods, I will refer to the documentation on MSDN.

Before addressing it, I will briefly recall an array, which also is a container. Consider as an example the following program:

```
using System;

namespace Exam44
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] t = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
         string[] s = new string[6];
         s[0] = "Svend";
         s[1] = "Valborg";
         s[2] = "Knud";
         s[3] = "Abelone";
         s[4] = "Valdemar";
         s[5] = "Gudrun";
         Array.Reverse(t);
         Array.Sort(s);
         foreach (int n in t) Console.Write("{0} ", n);
         Console.WriteLine();
         foreach (string v in s) Console.Write("{0} ", v);
         Console.WriteLine();
      }
   }
}
```


Download free eBooks at bookboon.com

If you run the program you get the following result:

There is not much mystery in it, and you should especially note the class *Array*, which make a number of static methods available to manipulate arrays.

In most cases, an array could be an alternative to a collection class, and the most collection classes are actually an encapsulation of an array. There is no problem in using arrays instead of collection classes, but conversely gives collection classes are great advantages. For example has an array a fixed size and it is up to the programmer to keep track of where each element is and whether there is room for it. Is there not room, it is necessary to create a larger array and copy the content of the old array to the new one. It is all that logic that is encapsulated in a collection class, and thus something that happens automatically behind the programmer's back. Besides that, the collection classes make different services available that fits typical applications.

As mentioned above, the collection classes are generic and thus have a strong type. That is, the objects as a concrete collection may contain, must be of a certain type. However, there is also a selection of non-generic collection classes:

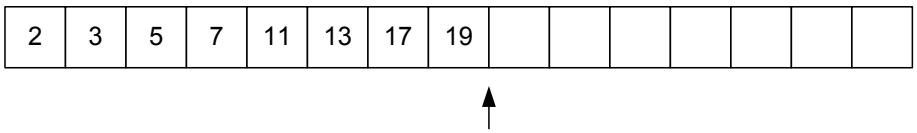
- ArrayList
- Stack
- Queue
- SortedList
- Hashtable

which are collections where objects have the type *object*. These classes are derived from the first versions of the .NET framework, which had not generic types available. The classes are still available and can be used, but it can be recommended to use the newer generic versions, since the result is a more stable code, because the compiler can test the type of the objects.

Download free eBooks at bookboon.com

24 List<T>

This type can best be interpreted as a dynamic array that is a type which in many ways can be used as an array, but may grow dynamically as needed. This is in contrast to an ordinary array, where you have to specify how many elements there must be room for when the array is created. One must not draw the comparison too far, and another interpretation is to think of the type as a sequence of elements, where you can always add elements to the end of the list. A *List* is a structure that at a given time has a capacity, and then you can add elements to the end of the list. The picture could be as shown below, where the capacity is 15, while there are 8 places that are used:



You can add elements where the arrow is pointing, and if at some point you exceed the capacity, it will automatically be extended. This is done by doubling the capacity.

Besides that you can add elements to the end of a *List*, the class offers a number of other methods, for example that one can delete an item, inserting an item in a certain place, etc. Internally, a *List* is an array, and it also means that you should be aware of the complexity of different methods. A *List* can not have empty places, and that means that if, for example you delete an item, all elements to the right of the deleted element must be moved one place to the left. Similarly, if you insert an element all the elements to the right of the location where the element is inserted, must be moved one place to the right to make room for the new element. In contrast the *Add()* that adds one element to the end of the list, is particularly effective since it can immediately place the element where the arrow is pointing – except in the situation where it is necessary to double the capacity.

Exam45

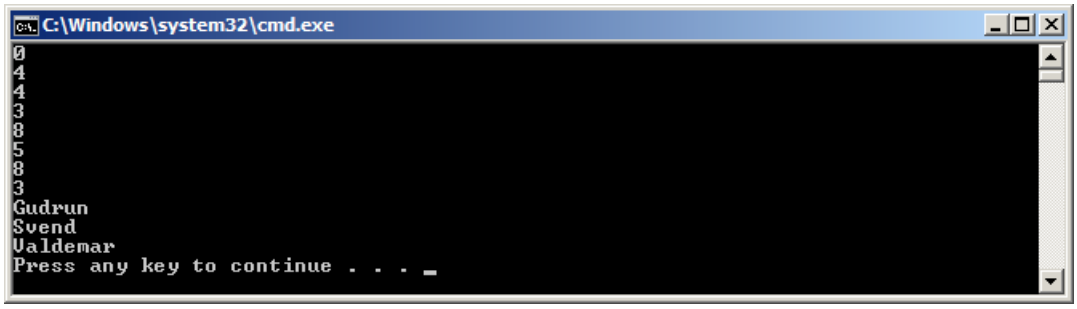
A List of strings

As an example the below program shows a method that creates a list of strings:

```
static void Main(string[] args)
{
   List<string> navne = new List<string>();
   Console.WriteLine(navne.Capacity);
   navne.Add("Svend");
   Console.WriteLine(navne.Capacity);
   navne.Add("Knud");
   navne.Add("Valdemar");
   Console.WriteLine(navne.Capacity);
   Console.WriteLine(navne.Count);
```


Download free eBooks at bookboon.com

```
    navne.Insert(0, "Olga");
    navne.Insert(0, "Gudrun");
    Console.WriteLine(navne.Capacity);
    Console.WriteLine(navne.Count);
    navne.Remove("Knud");
    navne.RemoveAt(1);
    Console.WriteLine(navne.Capacity);
    Console.WriteLine(navne.Count);
    for (int i = 0; i < navne.Count; ++i) Console.WriteLine(navne[i]);
}
```



Click on the ad to read more

Download free eBooks at bookboon.com

Explanation

First the program print the capacity of the list that at the start is 0. However, it is possible to create a list, using as a parameter to the constructor that sets the starting capacity. After the addition of an element, the program prints the capacity again. It is now 4, which means that the first time you add an item to the list it allocates space for 4 elements. After added two more elements and prints the capacity again and the number of elements, they are respectively 4 and 3, which is not surprising. As a next step the program insert two elements at the beginning of the list (in position 0) with the method *Insert()*. The capacity is then 8 (is doubled), and the number of elements is 5. Then the program deletes two elements, and the capacity is still 8. It should be noted that a list not automatically shrinks. The last for loop print all elements in the list, and the important thing here is to note that one can use the index operator.

Comment

Of the collection classes that are mentioned above, is a *List* the most commonly used and it is applied basically when you do not have the ability to estimate the size of an array.

Exam68

Enter sale of products

The aim of this example is to show a further application of a *List*.

The task is to write a program where the user can enter information about a product sale. The user must enter some information (records) consisting of a monthly number, and an amount and it is repeated until there is no more information. As an example the user could enter:

Month	Amount
5	800
8	1000
3	120
5	200
3	700
5	1500
....	

This means that the numbers can come in any order for months, and there may be several amounts for the same month.

After entering, the program must print a listing that for each month shows a line for each product sale and the total sale – that is the sum of all item amounts for the current month.

Download free eBooks at bookboon.com

How to

Basically, the task can be broken down into two sub-problems:

- Entering of sales items
- Print the result

A record consists of a monthly number and an amount, and I will therefore start with a type that can represent a single record:

```
struct Sale : IComparable<Sale>
{
   public int month;
   public double value;

   public int CompareTo(Sale sale)
   {
      return month.CompareTo(sale.month);
   }
}
```

As records come in any order, they must be stored somewhere, before the result can be counted together, and here I will use a *List*. Every time you enter a new record, I'll add it to the *List*. The list may be defined as follows:

```
static List<Sale> sales = new List<Sale>();
```

The first sub-problem can be described as:

```
repeat
{
      enter mounth number
      if month == 0 then terminate
      enter amount
      add a Sale object to the list
}
```

The entering routine can now be written in C# as:

```
static void AddSales()
{
   while (true)
   {
      Sale sale;
      sale.month = Enter.EnterInt("Enter 1, 2, ..., 12 for month or 0 for stop");
      if (sale.month == 0) break;
      if (sale.month >= 1 && sale.month <= 12)
      {
         sale.value = Enter.EnterDouble("Enter item amount");
         sales.Add(sale);
      }
      else Console.WriteLine("Ulovlig værdi for måned...");
   }
}
```


Download free eBooks at bookboon.com

Then there is the other sub-problem. The numbers should be printed in order of the months, so I will start by sorting the content of the *List* for months, so I know when the list is traversed that one comes to numbers in the right order. Then the task is in principle simple and mainly consists in summarizing month totals and print product amounts. The hardest part is to print the totals at the right time:

```
let sum = 0
as long as there are more records in the list repeat
{
   if the record is from the same month then
   {
      add the amount to sum
   }
   or
   {
      print the total for the month
      start with a new month
   }
   print the product amount
}
```

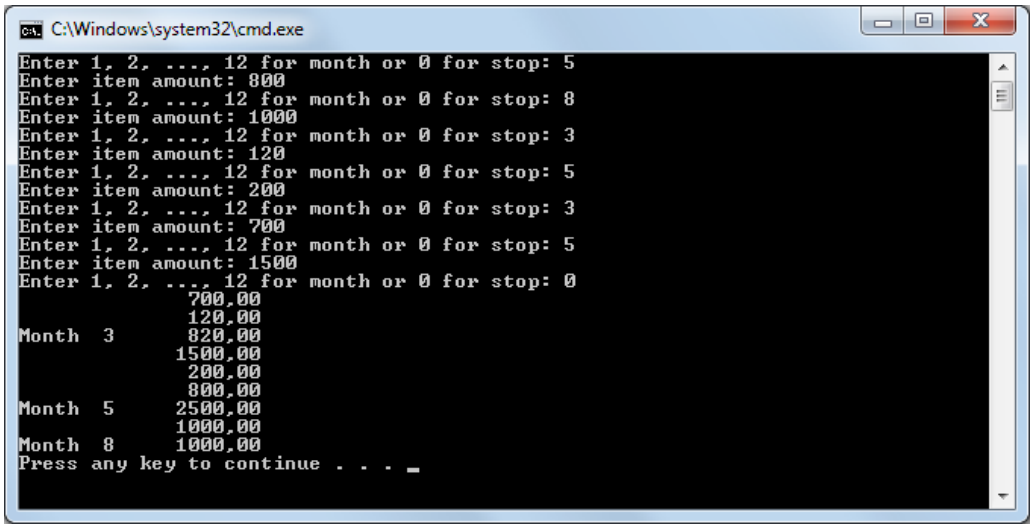
The algorithm can be written in C# as follows:

```
static void Print()
{
   int month = 0;
   double sum = 0;
   foreach (Sale sale in sales)
   {
      if (sale.month == month)
         sum += sale.value;
      else
      {
         if (month > 0) Console.WriteLine("Month {0, 2}{1, 12:F2}", month, sum);
         sum = sale.value;
         month = sale.month;
      }
      Console.WriteLine("{0, 20:F2}", sale.value);
   }
   Console.WriteLine("Month {0, 2}{1, 12:F2}", month, sum);
}
```

Then there is the *Main()* method, and here is the only one outstanding to sort the list, but the *List* class has a method for that purpose, that does it all, as long as the list's objects implements the *IComparable* interface, but that is precisely the case for the type of *Sale*:

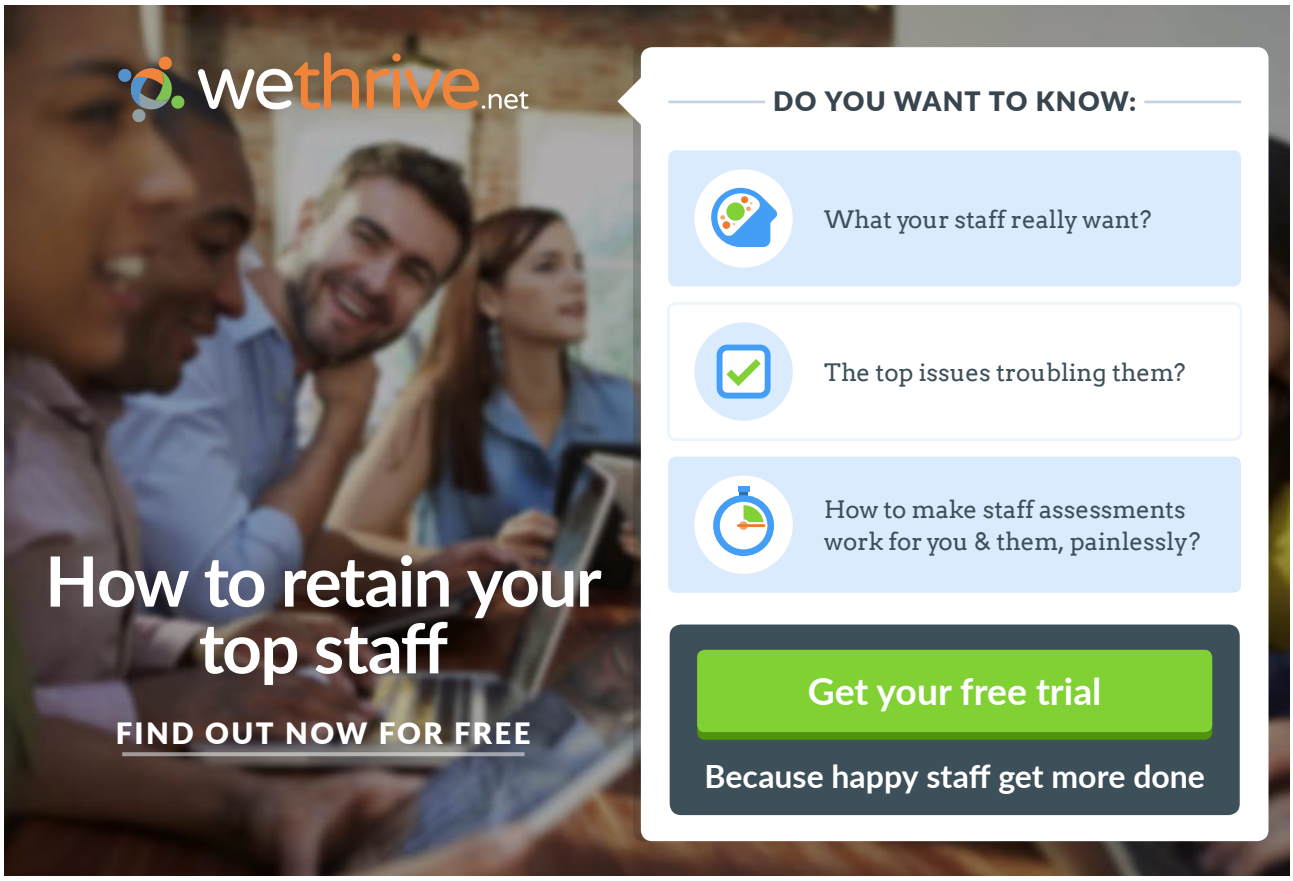
```
static void Main(string[] args)
{
   AddSales();
   sales.Sort();
   Print();
}
```


Download free eBooks at bookboon.com



Explanation

Regarding the type of *Sale* there is not much to explain beyond, once again noting that it implements the *IComparable* interface so that *Sale* objects can be compared and thus sorted. The type is a *struct* instead of a *class*, and there is no specific justification for that only once again to remind you of a *struct*, and to show that a struct can implement an interface. Although that in this connection there is no particular advantage of using a struct, then *Sale* indeed is a good candidate for a struct as the type consisting solely of two simple variables.



Click on the ad to read more

Download free eBooks at bookboon.com

The list itself is defined as a static variable in the class and you should primarily observe how the list is defined to contain Sale items. Also note how it is sorted in Main().

The most important in the method *AddSales()* is how to create *Sale* objects (as *Sale* is a struct, it is not necessary to create an object with new) and adds them to the list. Most of the method has to do with the input, using two input methods, which are not shown above. You can find them in the final code.

The method *Print()* has two local variables:

- *month* to keep track of the number of the current month
- *sum* to summing the total for a particular month

Otherwise, the method consists of a loop over the list's elements, and here it is important to keep track of whether there is a change of month. If applicable, the total is printed, and the two auxiliary variables must be initialized for a new month.

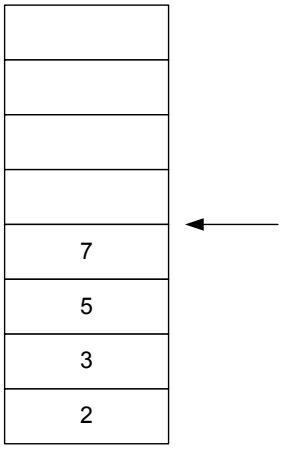
Download free eBooks at bookboon.com

25 Stack<T> and Queue<T>

A stack is a collection which has two basic operations

- one can put an item on the stack, an operation that is called *push*
- one can remove the element that was last placed on the stack, an operation that is called *pop*

For variables, I have several times mentioned the program stack. This is an example of a data structure that is a stack. You also sometimes call the data structure a LIFO (Last In First Out) structure, corresponding to that there only is access to the element that last are put on the stack. One think typical of the structure as a container where you can place an element at the top, where the arrow is pointing and where you can only remove the item that the arrow is pointing at – it stacks elements on top of each other and hence the name.



A concrete implementation of the type and that includes the type *Stack<T>* in C#, will usually also define other methods. For example a method to refer to the element at the top of the stack without removing it, and a method in order to test whether the stack is empty.

Download free eBooks at bookboon.com

Exam46

Stack of integers

As an example is shown a program that creates a stack of integers and places 8 numbers on the stack. Then the program empties the stack, and the numbers are printed on the screen:

```
static void Main()
{
   Stack<int> s = new Stack<int>();
   s.Push(2);
   s.Push(3);
   s.Push(5);
   s.Push(7);
   s.Push(11);
   s.Push(13);
   s.Push(17);
   s.Push(19);
   while (s.Count > 0) Console.WriteLine(s.Pop());
}
```

One should note that the numbers are printed in reverse order of how they are put on the stack – the number that was last placed on the stack is printed first.

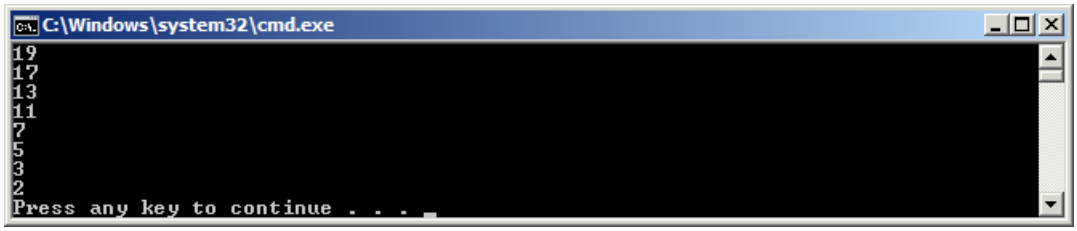
Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com



Exam69

StackSort

There are many uses of a stack, and in the book's last part I will show a typical application. In this example, I show how a stack may be used to sort an array.

How to

Given two stacks – hereinafter referred to as the left and the right stack – one can sort the array in the following manner:

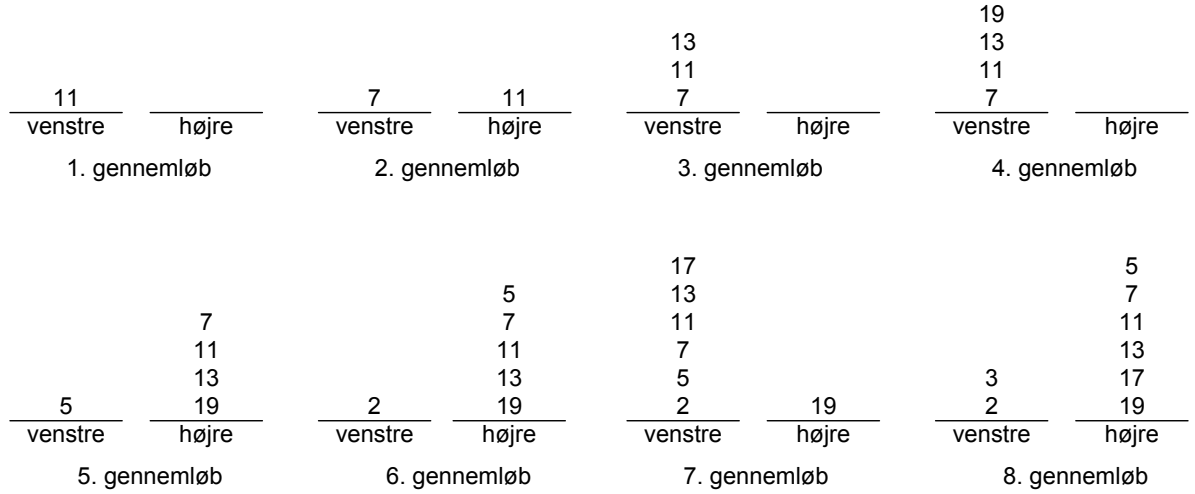
```
for each element t in the array repeat
{
   as long as t is less than the top of the left stack repeat
   {
      pop the left stack and push the element on the right stack
   }
   as long as t is greater than the top of the right stack repeat
   {
      pop the right stack and push the element on the left stack
   }
   push t on the left stack
}
as long as the left stack is not empty repeat
{
   pop the left stack and push the element on the right stack
}
loop over the array from start to end
{
   pop the right stack and insert the element in the array
}
```

It's not so easy to understand that the algorithm actually is a sort, and the easiest is to do a desktop test with a simple example. For this given an array with 8 elements:

11 7 13 19 5 2 17 3

Below are illustrated what happens with the two stacks when the array is traversed from left to right:

Download free eBooks at bookboon.com



If we empty the left stackt over on the right stack, it contains all the elements sorted in ascending order.

Below is shown how the algorithm can be implementet as a generic method in C#:

```
class Program
{
   static void Main(string[] args)
```

Click on the ad to read more

Download free eBooks at bookboon.com

```
    {
        int[] v = { 23, 17, 7, 19, 29, 2, 11, 5, 3, 13 };
        Print(v);
        Sort(v);
        Print(v);
    }

    static void Sort<T>(T[] arr) where T : IComparable<T>
    {
        Stack<T> s1 = new Stack<T>();
        Stack<T> s2 = new Stack<T>();
        foreach (T t in arr)
        {
            while (s1.Count > 0 && s1.Peek().CompareTo(t) > 0) s2.Push(s1.Pop());
            while (s2.Count > 0 && s2.Peek().CompareTo(t) < 0) s1.Push(s2.Pop());
            s1.Push(t);
        }
        while (s1.Count > 0) s2.Push(s1.Pop());
        for (int i = 0; i < arr.Length; ++i) arr[i] = s2.Pop();
    }

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

Explanation

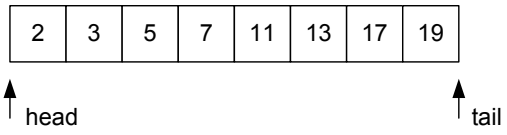
The algorithm is implemented by method *Sort()*. You should note that it is a generic method and that the parameter type is *IComparable*. It is necessary that the elements can be compared. Note also that the class *Stack* has a *Count* property that indicates the number of elements on the stack. All collection classes have this property. In this example the *Count* property is used to investigate whether the stack is empty. A stack has also a method which is called *Peek()*. It is used to refer to the element, which is located at the top of the stack, but without removing it.

Comment

The above sorting method is simple and takes up very little, but it is not very efficient, and is included here only as an example of the use of a stack and for the sake of the algorithm. The problem of the algorithm is that there are many movements of the elements between the two stacks.

Queue

If a stack is a LIFO data structure, a queue is a FIFO (First In First Out) data structure in which it is always the first (the oldest) element which is removed from the queue. The picture of a queue is something like the following



Download free eBooks at bookboon.com

where one inserts (adds to the queue) at the place as *tail* arrow points, and removes from the queue at the place as *head* arrow is pointing. The class *Queue<T>* represents a queue, and the two basic operations for inserting and removing elements is called respectively *Enqueue()* and *Dequeue()*. As an example is shown a program that creates a queue:

```
static void Main()
{
   Queue<int> q = new Queue<int>();
   q.Enqueue(2);
   q.Enqueue(3);
   q.Enqueue(5);
   q.Enqueue(7);
   q.Enqueue(11);
   q.Enqueue(13);
   q.Enqueue(17);
   q.Enqueue(19);
   while (q.Count > 0) Console.WriteLine(q.Dequeue());
}
```

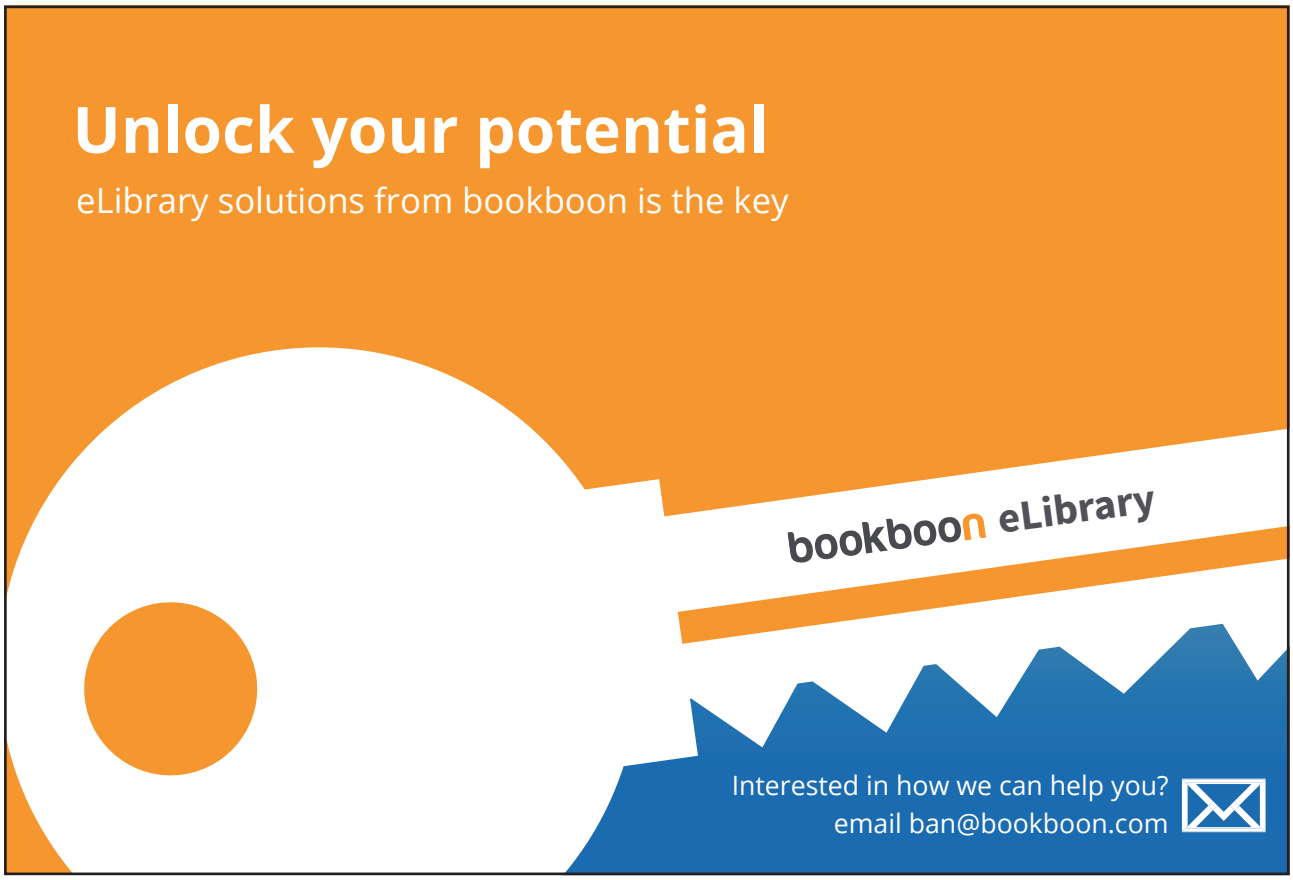
In practice there are not so many applications of a queue as a stack.

Download free eBooks at bookboon.com

26 LinkedList<T>

Above I have discussed the type *List<T>* as a sequence of elements. Its main feature is that you can add items to it and it will expand as needed, and it is even the case that adding elements to the end of the list is very effective. But it does have a few drawbacks. Firstly, the insertion or deletion of elements in the middle of the list is not particularly effective, since all the remaining elements have to be moved. Second, it is that the list is expanding by doubling mean that there can be a great space wastage and it takes time to create a new array and copy the contents of the old to the new. Consider as an example the following method:

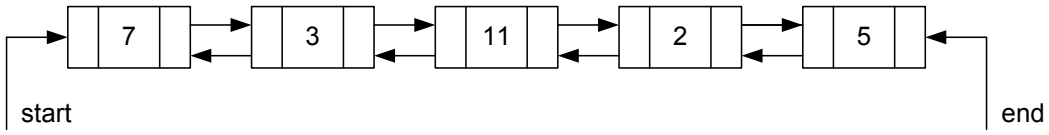
```
static void Main()
{
   List<int> list = new List<int>();
   for (int i = 0; i <= 1048576; ++i) list.Add(i);
   Console.WriteLine(list.Capacity);
   Console.WriteLine(list.Count);
}
```



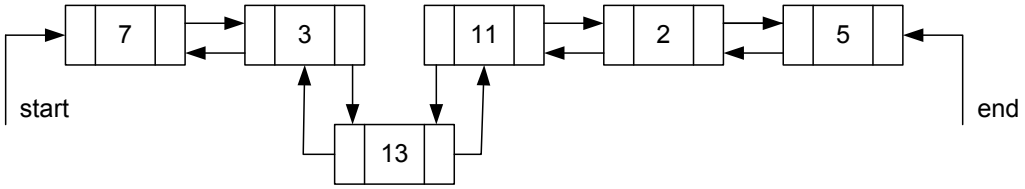
Click on the ad to read more

Download free eBooks at bookboon.com

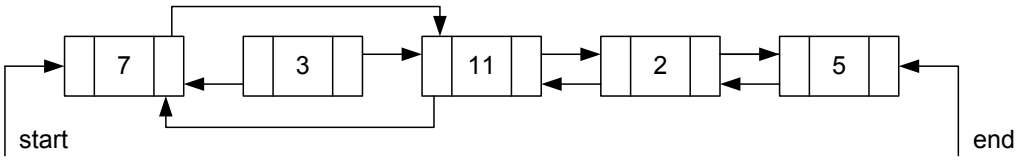
The list contains the elements 1048577, but also has 1048575 slots that are not used. This is where a *LinkedList* enters the picture as a list that allocates memory in another way. The idea is that, for each element there are attached two references, which can refer to, respectively the foregoing and the following element. One can think of a linked list as the following structure



where there is a reference to the start of the list and a reference to the end of the list. Each element (box) in the list is usually called a node. This structure solves the problem of insertion and deletion, as you can insert a new element in the middle of the list by simply changing some references, and in the same way you can remove an item. The figure below shows how it looked, if you insert an element with a value of 13 between 3 and 11 – it is necessary to change 4 references:



Below is in the same manner shown how it looks if you remove the 3th element. It is necessary to change two references, and there are now no references to the 3th element, so it automatic is destroyed by the grabage collector.



On the other hand, the problem of space consumption is not solved. There is no longer allocated space for items that are not used, but to each element there is attached two additional references, which also takes place.

A linked list solves the problem of insertion and deletion in the middle of the list, but one can not directly refer to the individual elements with an index. There is a need to seek from the beginning (or end) of the list until you find the item you are interested in.

The class *LinkedList* implements a linked list, and it differs from the class *List* of having other methods.

Download free eBooks at bookboon.com

Exam48

LinkedList of names

Below is shown how to create a *LinkedList* and place some elements in it:

```
static void Main(string[] args)
{
   LinkedList<string> list = new LinkedList<string>();
   list.AddLast("Svend");
   list.AddLast("Knud");
   list.AddLast("Valdemar");
   list.AddAfter(list.Find("Knud"), "Karlo");
   list.AddBefore(list.Find("Karlo"), "Frede");
   list.AddFirst("Gudrun");
   Print1(list);
   list.AddFirst("Olga");
   list.AddFirst("Abelone");
   Print2(list);
   Print3(list);
}

static void Print1(LinkedList<string> list)
{
   foreach (string name in list) Console.WriteLine(name);
   Console.WriteLine("-----------------");
}

static void Print2(LinkedList<string> list)
{
   for (LinkedListNode<string> node = list.Last; node != null; node = node.Previous)
   Console.WriteLine(node.Value);
   Console.WriteLine("-----------------");
}

static void Print3(LinkedList<string> list)
{
   for (int i = 0; i < list.Count; ++i) Console.WriteLine(list.ElementAt(i));
   Console.WriteLine("-----------------");
}
```

Explanation

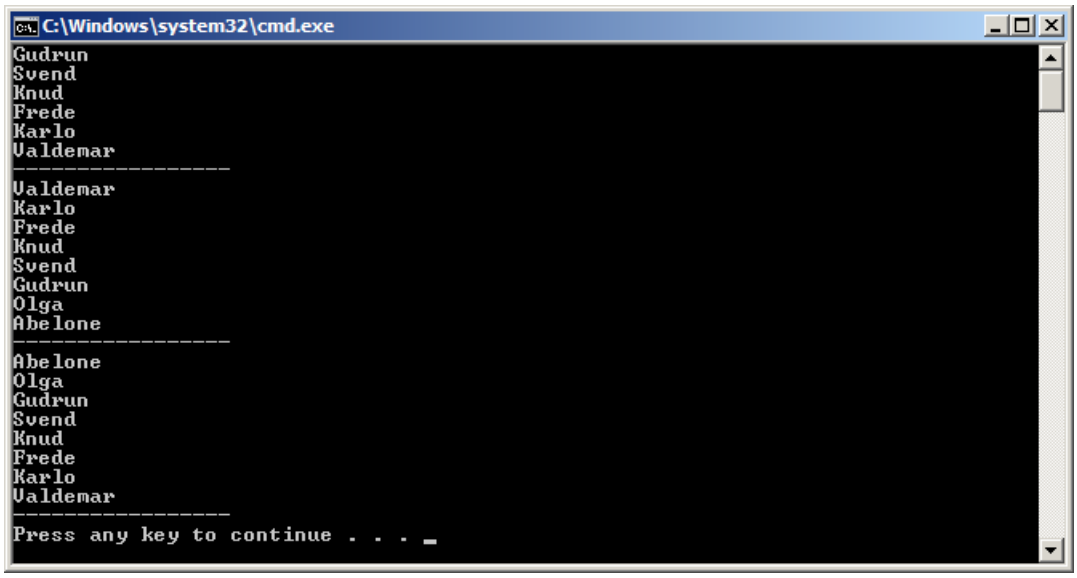
There are four ways to insert an element

1. Insert it at the end of the list
2. Insert it at the beginning of the list
3. Insert it after an existing element
4. Insert it before an existing element

You should note how the method *Find()* finds the element to be inserted relative to. It returns a *LinkedListNode*, which is a structure similar to a node.

Download free eBooks at bookboon.com

Note the three *Print()* methods. The first there is not much to tell, since it prints the list with a normal *foreach* loop. The second method use a property to obtain a reference to the last node – note again the type *LinkedListNode*. Then the list is traversed from behind. In a *LinkedList* you can not reference the elements with an index, but there is a method *ElementAt()*, which gives the same result. It is shown in the last printing method. Note that *ElementAt()* really counts from the beginning of the list and forward, so it's not a fair solution to the printing problem.



Comment

Whether to use a *LinkedList* or a *List* is determined by the task, since both data structures have their advantages and disadvantages. There is no doubt that a *List* is used more often.

Download free eBooks at bookboon.com

27 Dictionary<K,V> and SortedDictionary<K,V>

A dictionary is a collection where each element is identified by a key. Therefore, also sometimes one refers to a dictionary as a collection of key / value pairs, where each value is identified by a key. You could compare a dictionary with a list, because in a list, the elements are identified by an index. When you have to find an item in the list, you not search for the item, but its position is calculated from the index. In the same way it is with a dictionary, when you need to find an item, then one calculates the element's position based on the key value – you do not search for the item. The following figure can be used to illustrate some of the technique, which stores key / value pair as zip code and city:

There needs a function that convert a zip code to a place in the container. Such a function is usually called a *hash* function, and instead of a dictionary you often refer to the structure as a hash table. In this case, the hash function should be simple, and consists simply in taking the last digit. For example are (9492, Blokhus) stored at position 2, since the zip code ends with 2. Similarly (8765, Klovborg) are stored in position 5, and (8883, Gjern) are stored on position 3.

The above is obviously very simplistic, but it illustrates two very fundamental problems:

- What to do if the structure is filled out and needs to be expanded – above, there's only room for 10 elements.
- What to do if there will be a collision – with the above hash function, (8543, Hornslet) are stored in the same place as Gjern.

Download free eBooks at bookboon.com

I shall not here explains, how to solve these problems, but just mention that it's problems that an implementation of the structure must necessarily solve.

The class *Dictionary* is a collection class to a dictionary or a hash table.

Exam49

Table of job titles

Below is an example where the key is a name, while the value is a job title:

```
static void Main()
{
   Dictionary<string, string> map = new Dictionary<string, string>();
   map.Add("Knud", "Konge");
   map.Add("Gudrun", "Heks");
   map.Add("Svend", "Kriger");
   map.Add("Olga", "Spåkone");
   map.Add("Valdemar", "Skarpretter");
   map.Add("Abelone", "Klog kone");
   Print1(map);
   map["Gudrun"] = "Sin mands kone";
   Print1(map);
   Print2(map);
   Print3(map);
}

static void Print1(Dictionary<string, string> map)
{
   for (int i = 0; i < map.Count; ++i) Console.WriteLine(map.ElementAt(i));
   Console.WriteLine("-----------------");
}

static void Print2(Dictionary<string, string> map)
{
   foreach (string name in map.Keys) Console.WriteLine(map[name]);
   Console.WriteLine("-----------------");
}

static void Print3(Dictionary<string, string> map)
{
   foreach (string job in map.Values) Console.WriteLine(job);
   Console.WriteLine("-----------------");
}
```

Explanation

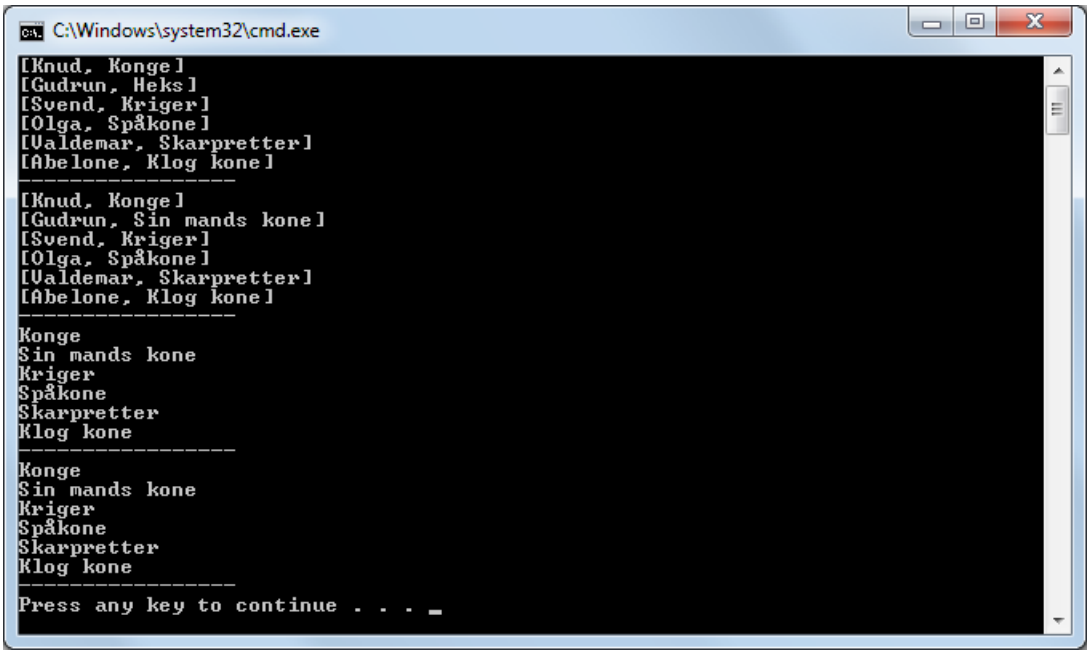
Note first how to add values to a dictionary. Note then the first print method and how to print the key / value pairs. The keys must be unique, and if you try to add a value with an existing key, you get an exception. However, you can change a value using the key as the index:

```
map["Gudrun"] = "Sin mands kone";
```

Note finally the last two printing methods, where the first method runs over all keys and the other runs over all values. Here you should note how to get a collection of all keys (using *Print2()*) and a collection of all values (using *Print3()*).

Download free eBooks at bookboon.com

If you execute the method you get the following result:



Click on the ad to read more

Download free eBooks at bookboon.com

Exam50

User defined key

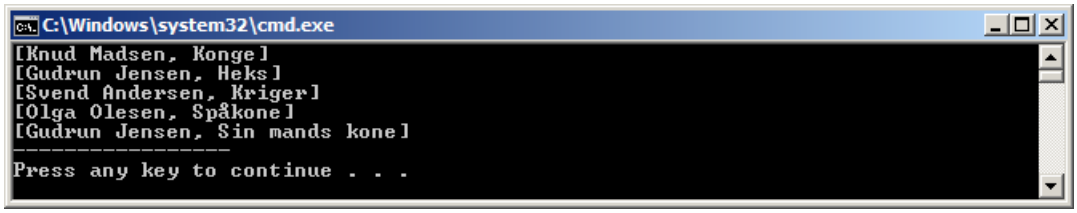
In a dictionary the type for both key and value may be anything – almost. Consider the following type:

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

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

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

Consider then a dictionary that has the type *Name* as key:

```
static void Main(string[] args)
{
   Dictionary<Name, string> map = new Dictionary<Name, string>();
   map.Add(new Name("Knud", "Madsen"), "Konge");
   map.Add(new Name("Gudrun", "Jensen"), "Heks");
   map.Add(new Name("Svend", "Andersen"), "Kriger");
   map.Add(new Name("Olga", "Olesen"), "Spåkone");
   map.Add(new Name("Gudrun", "Jensen"), "Sin mands kone");
   Print(map);
}

static void Print(Dictionary<Name, string> map)
{
   for (int i = 0; i < map.Count; ++i) Console.WriteLine(map.ElementAt(i));
   Console.WriteLine("-----------------");
}
```

Note that the program can be translated and executed, but the result is not as expected as key *Gudrun Jensen* occurs twice:



Download free eBooks at bookboon.com

The reason is that the type *Name* not overrides *GetHashCode()*. As mentioned builds a dictionary of a hash function to calculate the individual elements position, which in turn uses this type's *GetHashCode()* method. The key type must then always override *GetHashCode()*:

```
public class Name
{
   ...

   public override int GetHashCode()
   {
      return firstname.GetHashCode() + lastname.GetHashCode();
   }
}
```

Exam51

A sorted dictionary

A dictionary generally guarantees nothing about how the elements are arranged and in which order they are meet at a traversal of the structure. In contrast, a *SortedDictionary* is a dictionary, where the elements are sorted by key order:

```
static void Main()
{
   SortedDictionary<string, string> map = new SortedDictionary<string, string>();
   map.Add("Knud", "Konge");
   map.Add("Gudrun", "Heks");
   map.Add("Svend", "Kriger");
   map.Add("Olga", "Spåkone");
   map.Add("Valdemar", "Skarpretter");
   map.Add("Abelone", "Klog kone");
   foreach (string navn in map.Keys) Console.WriteLine("{0}, {1}", navn, map[navn]);
}
```

That is, the result is ordered by the key which is *name* and note that this is not the same order as objects are added to the structure.

Download free eBooks at bookboon.com

Exam52

Comparable keys

The following can't be executed (you get an exception):

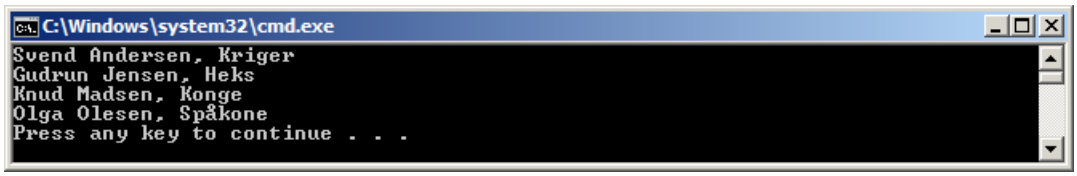
```
static void Test11()
{
   SortedDictionary<Name, string> map = new SortedDictionary<Name, string>();
   map.Add(new Navn("Knud", "Madsen"), "Konge");
   map.Add(new Navn("Gudrun", "Jensen"), "Heks");
   map.Add(new Navn("Svend", "Andersen"), "Kriger");
   map.Add(new Navn("Olga", "Olesen"), "Spåkone");
   foreach (Navn navn in map.Keys) Console.WriteLine("{0}, {1}", navn, map[navn]);
}
```

In order that the keys can be sorted, they must be comparable and it will say that the key type must implement the *IComparable* interface:

```
public class Name : IComparable<Name>
{
   ...

   public int CompareTo(Name name)
   {
      if (lastname.CompareTo(name.lastname) == 0)
         return firstname.CompareTo(name.firstname);
      return lastname.CompareTo(name.lastname);
   }
}
```

Click on the ad to read more
Download free eBooks at bookboon.com

Here the keys is sorted so that the keys are first compared on the last name, and if the last names are the same, the keys are compared on the first name.



Comment

A *SortedDictionary* is internally a binary search tree, and without in this place to elaborate on what it is, can I mention that it is a structure that optimizes the search of elements. There are actually another collection class

SortedList<K, V>

that seen from the programmer is the same as a *SortedDictionary*, but also internally is a binary search tree. The difference is that a *SortedList* uses less memory than a *SortedDictionary*, and vice versa is a *SortedDictionary* more efficient at inserting and deleting of items.

Exam70

Cue list

In this example I will show how to represent a cue list for a book. The purpose is mainly to show how to use a *SortedDictionary*, but also a *LinkedList*.

A cue consists of a name and a number of page references where a name is a string, while a page reference is an integer and a cue list is a list of items of that kind. The task is to write a program that can create a cue list and print out the list on the screen.

How to

I will start by defining a class that can represent an element to the list consisting of a name and page references:

```
public class PageReferences
{
   private string name;
   private LinkedList<int> list = new LinkedList<int>();

   public PageReferences(string name, int page)
   {
      this.name = name;
      list.AddFirst(page);
   }
```


Download free eBooks at bookboon.com

```
   public string Name
   {
      get { return name; }
   }

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

   public int this[int n]
   {
      get { return list.ElementAt(n); }
   }

   public int[] References
   {
      get { return list.ToArray(); }
   }

   public void Add(int page)
   {
      for (LinkedListNode<int> node = list.First; node != null; node = node.Next)
         if (node.Value == page) return;
         else if (node.Value > page)
         {
            list.AddBefore(node, page);
            return;
         }
      list.AddLast(page);
   }

   public override string ToString()
   {
      StringBuilder builder = new StringBuilder(name + ",");
      for (LinkedListNode<int> node = list.First; node != null; node = node.Next)
         builder.Append(" " + node.Value);
      return builder.ToString();
   }
}
```

I will keep the key words in a *SortedDictionary* with *name* as the key. This provides two advantages:

- The list can be traversed, and thus printed sorted by name
- one can find siderefenrencer to a particular cue without searching, but by direct lookup

When the keys are compared, I'm not interested in making a distinction between uppercase and lowercase letters. Therefore I have written my own key type:

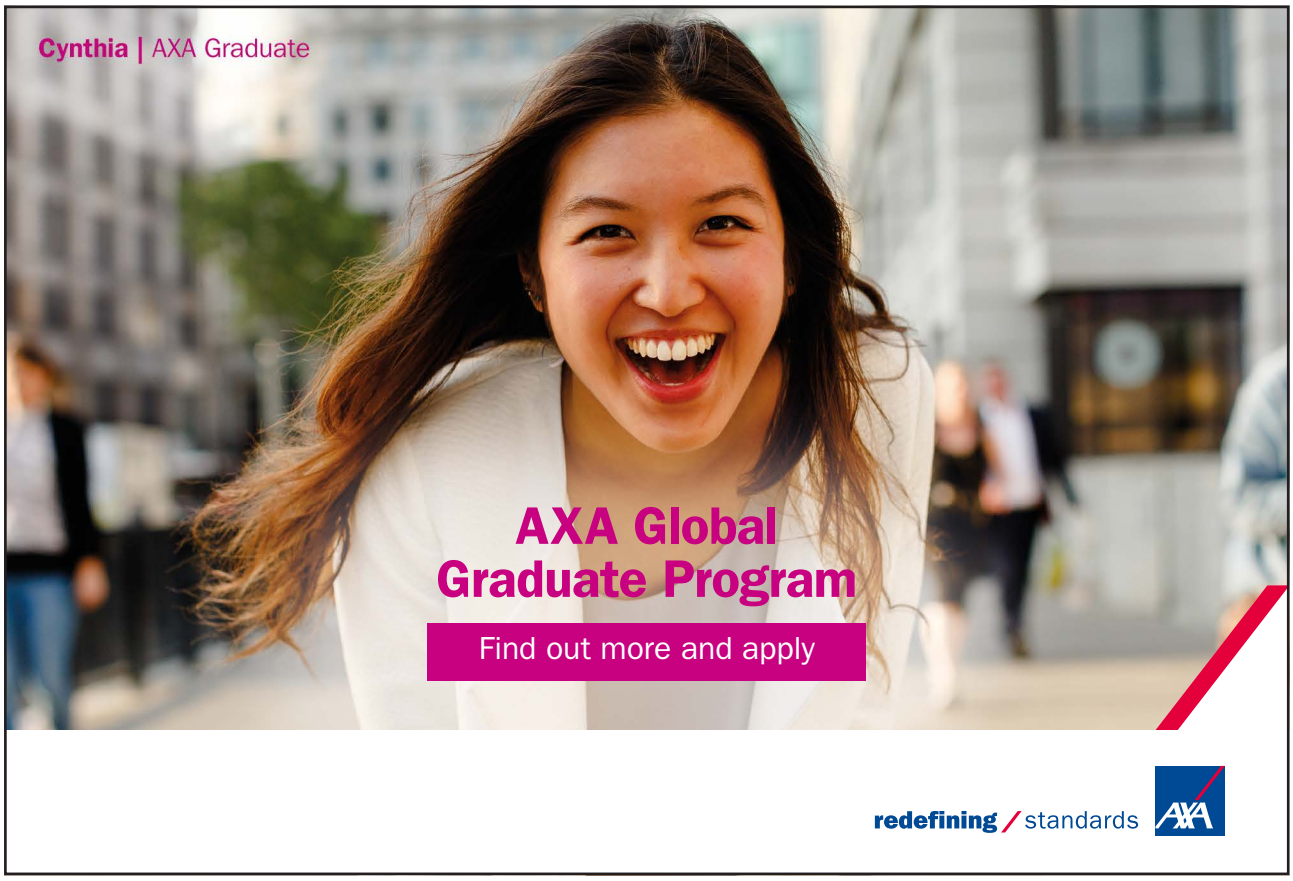
```
class PageKey : IComparable<PageKey>
{
   private string name;

   public PageKey(string name)
   {
      this.name = name;
   }
```


Download free eBooks at bookboon.com

```
   public string Value
   {
      get { return name; }
   }

   public override bool Equals(object obj)
   {
      if (!(obj is string)) return false;
      return name.Equals((string)obj, StringComparison.CurrentCultureIgnoreCase);
   }

   public override int GetHashCode()
   {
      return name.ToUpper().GetHashCode();
   }

   public int CompareTo(PageKey key)
   {
      return string.Compare(name, key.name, true);
   }
}
```

After that the class to the cue list can be written:

```
class Index
{
   private SortedDictionary<PageKey, PageReferences> table =
      new SortedDictionary<PageKey, PageReferences>();
```



Click on the ad to read more

Download free eBooks at bookboon.com

```
   public bool Add(string name, int page)
   {
      name = name.Trim();
      if (name.Length == 0) return false;
      PageKey key = new PageKey(name);
      if (table.ContainsKey(key)) table[key].Add(page);
      else table.Add(key, new PageReferences(name, page));
      return true;
   }

   public int[] Pages(string name)
   {
      PageKey key = new PageKey(name);
      if (table.ContainsKey(key)) return table[key].References;
      else return new int[0];
   }

   public void Print()
   {
      char ch = ' ';
      foreach (PageKey key in table.Keys)
      {
         PageReferences pages = table[key];
         if (ch != char.ToUpper(pages.Name[0]))
         {
            ch = char.ToUpper(pages.Name[0]);
            Console.WriteLine(ch);
         }
         Console.WriteLine(pages);
      }
   }
}
```

Explanation

There is this many things to note. If starting with the class *PageReferences*, note that the list of page references is a *LinkedList*. The reason is that I want to insert page numbers in ascending order, which happens in the *Add()* method. The list is created in the constructor, and when the *Add()* method is performed, the list will always contain at least one page reference. The *Add()* method traverses the page references for the current cue. If the page reference is already there, there is nothing more than the insertion is ignored (the same page reference should not occur more than once). If you find a page reference that is greater than the new reference, it should be inserted before the element that you have reached, and we have thus obtained the list of page references are sorted. If you reach through the list without the new reference was inserted, it is because the reference should to be inserted as the last reference in the list.

Besides the method *Add()*, the class *PageReferences* has different properties. Note in particular that there is an override of the index soperator (which are not used in the example). Note also the property *References*, which returns an array with page numbers. This array is formed with the method *ToArray()* in the class *LinkedList*. All the collection classes have this method.

Download free eBooks at bookboon.com

The class *PageKey* defines the key as an encapsulation of a *string*. Notice how the class implements the *IComparable* interface, so the comparison does not distinguish between uppercase and lowercase letters (it indicates the last parameter). Note also that *Equals()* does not distinguish between uppercase and lowercase letters, and that *GetHashCode()* is also implemented.

Finally, there is a class *Index*, which represents key words from the list. The class provides three services available:

- You can add new keywords to the list that either insert a new cue, or update an existing one.
- You can get an array with page references to a particular cue, and here you must particularly note the syntax for how to create an empty array if the cue does not exist.
- You can print the list on the screen, which in principle is merely a matter to traverse the list in the order in which the keys specify.

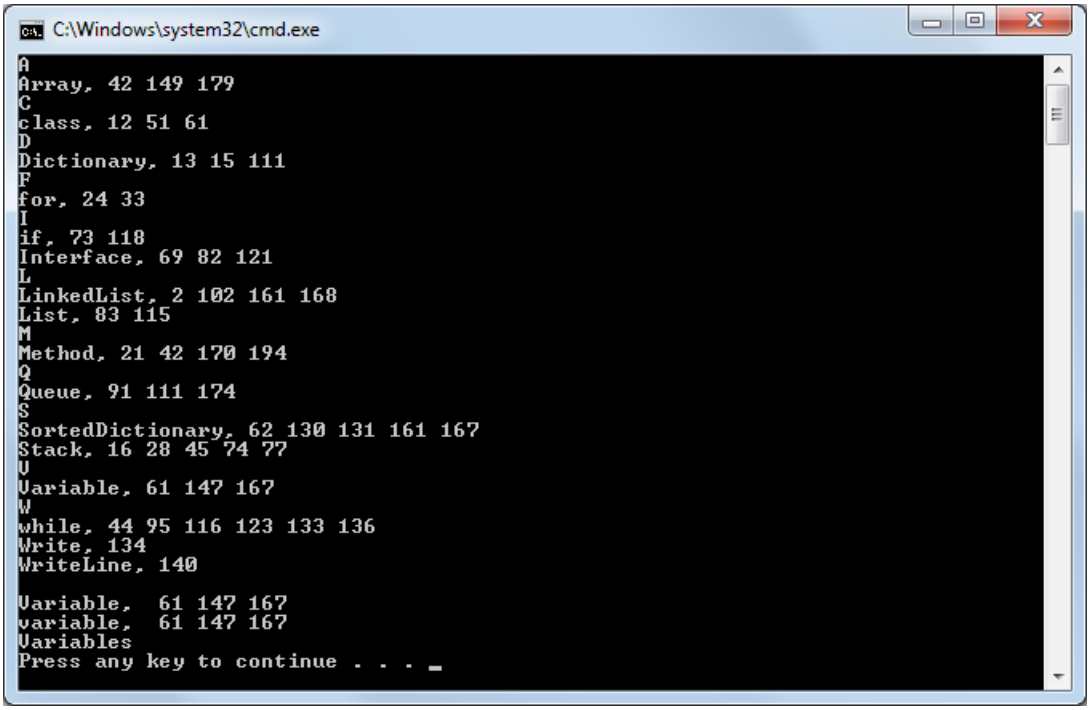
Test

Finally, I show a test program:

```
class Program
{
   static Random rand = new Random();
   static string[] words = { "Write", "if", "for", "while", "Variable", ..... };

   static void Main(string[] args)
   {
      Index index = new Index();
      for (int i = 0; i < 50; ++i)
         index.Add(words[rand.Next(words.Length)], rand.Next(200));
      index.Print();
      Console.WriteLine();
      Print("Variable", index.Pages("Variable"));
      Print("variable", index.Pages("variable"));
      Print("Variables", index.Pages("Variables"));
   }

   static void Print(string name, int[] pages)
   {
      Console.Write(name);
      if (pages.Length > 0)
      {
         Console.Write(", ");
         for (int i = 0; i < pages.Length; ++i) Console.Write(" " + pages[i]);
      }
      Console.WriteLine();
   }
}
```

In the beginning of the program there is defined a few words. These are then inserted randomly into a cue list with random page references:

Download free eBooks at bookboon.com



FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Arriving 33
Living 50
Studying 51
Working 101
Research 50

Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

VISIT FACTCARDS.NL

Click on the ad to read more

Download free eBooks at bookboon.com

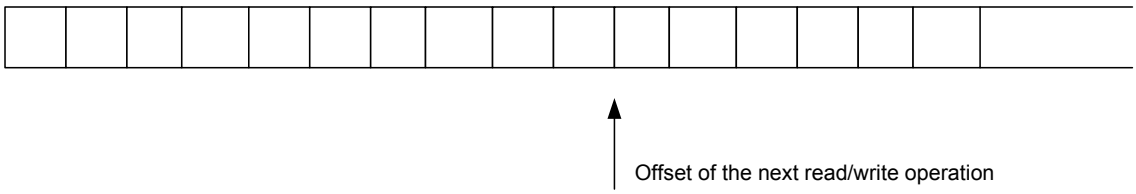
Part 4 IO

Previously, files was a central concept in programming, and even if it does not play the same role as before, it is still important to know how to write to and read from a file.

When IO and files does not play the same role as before, it is because that applications that need to manipulate persistent data stored on disk usually use database products, and anything relating to access to files and access to data is thus left to the current database system. Nevertheless, there are still situations where there is a need to be able to access regular files, and it is mainly in two situations: How to read and write a plain text file, and how to serialize and deserialize an object.

As everywhere in .NET is access to files relieved the programmer through a number of finished classes, which are primarily found in the *System.IO* namespace. It is a comprehensive namespace with many classes, but fortunately it is not necessary to know them all, so I will concentrate on those that as a minimum is required.

From a program a file is basically a sequence of bytes, and in many ways one can think of a file as a 1-dimensional array with no upper limit (the size is obviously limited by the space available on disk). The place where the next file operation (reading or writing) is performed is determined by a so-called file pointer, which is really just a counter relative to the strat of the file. Correspondingly, one can illustrate a file in the following manner, wherein each box is a byte:



In a program, the challenge is to control where to read from and write to the file and to interpret the result that is readed and convert it to the correct data format. It's actually all that the classes in the IO system and their methods encapsulate.

Download free eBooks at bookboon.com

28 Text files

A text file is a file containing a number of lines, where each line is a string. Since the strings in .NET are represented as unicode sequences of characters, each character is written as 2 bytes to the file, but it is enclosed by IO system. Text files are still important. For example text files are often used for exchanging data in the form of comma delimited files or similar. One should also note that, for example HTML files or XML files are text files.

Exam51

Write and read text

I will write a program that shows how to write and read a simple text file. When the program runs, the user must enter text lines, and they are saved in a file. The entry is terminated when the user simply presses Enter, and the program will then read the file and print the content on the screen.

How to

To create a text file and write to it can be described as the following algorithm:

```
open/create the file
get data
as long as there are data
   write a line in the file
   get data
close the file
```

and the important thing is that it is actually happening in the same way every time, and it is just the operation *get data* which may vary. In the same way, reading a text file can be written as the following algorithm:

```
open the file
read a line
as long as there is a line
     use the line
     read a line
close the file
```

and here it is again important to note that the procedure is the same regardless of the content of the file, and only the operation use the line varies.

The program can be written as follows:

```
using System;
using System.IO;
```


Download free eBooks at bookboon.com

```
namespace Exam53
{
   class Program
   {
   static void Main(string[] args)
   {
      Write();
      Read();
   }

   public static string GetLine()
   {
      Console.Write("? ");
      return Console.ReadLine();
   }

   public static void PutLine(string line)
   {
      Console.WriteLine(line);
   }

   public static void Write()
   {
      StreamWriter writer = new StreamWriter("Tekst.txt");
      string line = GetLine();
      while (line.Length > 0)
      {
         writer.WriteLine(line);
         line = GetLine();
      }
      writer.Close();
   }
```

Click on the ad to read more

Download free eBooks at bookboon.com

```
   public static void Read()
   {
         StreamReader reader = new StreamReader("Tekst.txt");
         string line = reader.ReadLine();
         while (line != null)
         {
            PutLine(line);
            line = reader.ReadLine();
         }
         reader.Close();
      }
   }
}
```

Explanation

Note first that there is a *using* statement for *System.IO*. It will typically be the case in all applications that use files.

The method *GetLine()* corresponds to *get data* in the algorithm, and it writes a simple prompt in the form of a question mark on the screen and then performing a *ReadLine()*. The method returns a *string* with the user's input.

The method *Write()* creates and opens a text file for writing:

```
StreamWriter writer = new StreamWriter("Tekst.txt");
```

The file is named *Tekst.txt* and is created in the same directory as the program's exe file. The file name is a parameter to the constructor for a class of the type *StreamWriter*, which represents a text file which you can write to. You can specify a full path to the file if you want it to be placed somewhere else. After the user enters a line, it is written to the file using the writer and the method *WriteLine()*:

```
writer.WriteLine(line);
```

Note that this method has the same properties (the same overrides) as the corresponding method in the *Console* class and, therefore, also can print a formatted text to a file.

After the loop is terminated, the file is closed.

There are a few things you should be aware of. When the file is opened by creating a *StringWriter* object, the file is created if it does not exist. If the file however, has already been created, it will be overwritten. If you do not want the file is overwritten, you can specify a second parameter to the constructor, which opens the file in *append* mode.

Download free eBooks at bookboon.com

There is then added to the end of the file. When you write a line in the file with the *WriteLine()*, the string's characters are written to the file followed by a newline. Finally, you should notice the last *Close()* statement. It is necessary because in fact the data are not written physical to the file for each *WriteLine()*, but instead to a buffer in the memory. *Close()* causes, in addition to free the connection to the file, that the content of the buffer is written to the file.

The method *Read()* shows how to read the content of a text file and print it on screen. Here corresponds the method *PutLine()* to apply the algorithm's *use line*, and in this case the method is tedious and do nothing more than to write the string as a line on the screen. The method *Read()* starts to open the file for reading with a *StreamReader* object:

```
StreamReader reader = new StreamReader("Tekst.txt");
```

Then the method trys to read a line in the file:

```
string line = reader.ReadLine();
```

If it goes well – *line* is different from *null* – the method *PutLine()* is executed with *line* as a parameter and then the program attempts to read the file again. The result is that the loop is repeated until there are no more data in the file. Finally, the file is closed.

This is actually what there is to say about text files, but it is clear that how to *get data*, and how to treat a line that is read from the file depends on the specific task.

Exam54

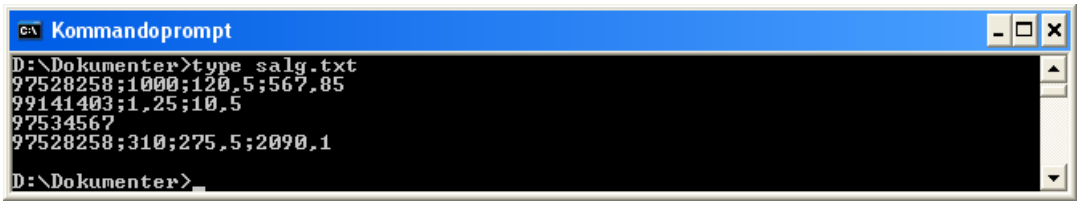
Write a comma separated file

The following example is in principle identical to the above, but the treatment of data is not quite so simple, and it is also a slightly more realistic example of how you can meet text files. Actually, there are two programs, and the next program is concerning the same example.

A comma separated file is a text file that contains a number of lines where each line consists of several elements, and where the individual elements are separated by a delimiter. The delimiter will often be a comma (hence the name) but it can in principle be anything, just to be a symbol that does not occur in the individual text elements. In this case a semicolon is used as a delimiter.

Download free eBooks at bookboon.com

The first program will create a text file where the user enters a phone number followed by one or more product amounts as you can think of sales and thus indicate sales to a particular phone number. For each phone number the program writes a line in the file consisting of the telephone number and product amounts and with the fields separated by semicolons. The result could be the following file:



Note that the same telephone number might well appear several times, and that there may be lines with just a phone and no amounts.

The code is as follows:

```
class Program
{
   static void Main(string[] args)
   {
      StreamWriter writer = new StreamWriter("F:\\Temp\\Salg.txt", true);
      string line = PhoneNumber();
      while (line.Length > 0)
```

Click on the ad to read more
Download free eBooks at bookboon.com

```
      {
         writer.WriteLine(line);
         line = PhoneNumber();
      }
      writer.Close();
   }

   static string PhoneNumber()
   {
      while (true)
      {
         Console.Write("Enter phone no. as 8 digits: ");
         string phone = Console.ReadLine().Trim();
         if (phone.Length == 0) return "";
         if (PhoneOk(ref phone))
         {
            StringBuilder builder = new StringBuilder(phone);
            double amount = Amount();
            while (amount > 0)
            {
               builder.Append(string.Format(";{0:F2}", amount));
               amount = Amount();
            }
            return builder.ToString();
         }
         Console.WriteLine("Illegal phone no.");
      }
   }

   static bool PhoneOk(ref string tlfnr)
   {
      tlfnr = tlfnr.Trim();
      StringBuilder builder = new StringBuilder();
      for (int i = 0; i < tlfnr.Length; ++i)
         if (tlfnr[i] >= '0' && tlfnr[i] <= '9')
            builder.Append(tlfnr[i]);
         else if (tlfnr[i] != ' ')
            return false;
         tlfnr = builder.ToString();
         return tlfnr.Length == 8;
   }

   static double Amount()
   {
      while (true)
      {
         Console.Write("Enter the amount: ");
         try
         {
            string text = Console.ReadLine().Trim();
            if (text.Length == 0) return 0;
            return Convert.ToDouble(text);
         }
         catch
         {
            Console.WriteLine("Illegal amount");
         }
      }
   }
}
```

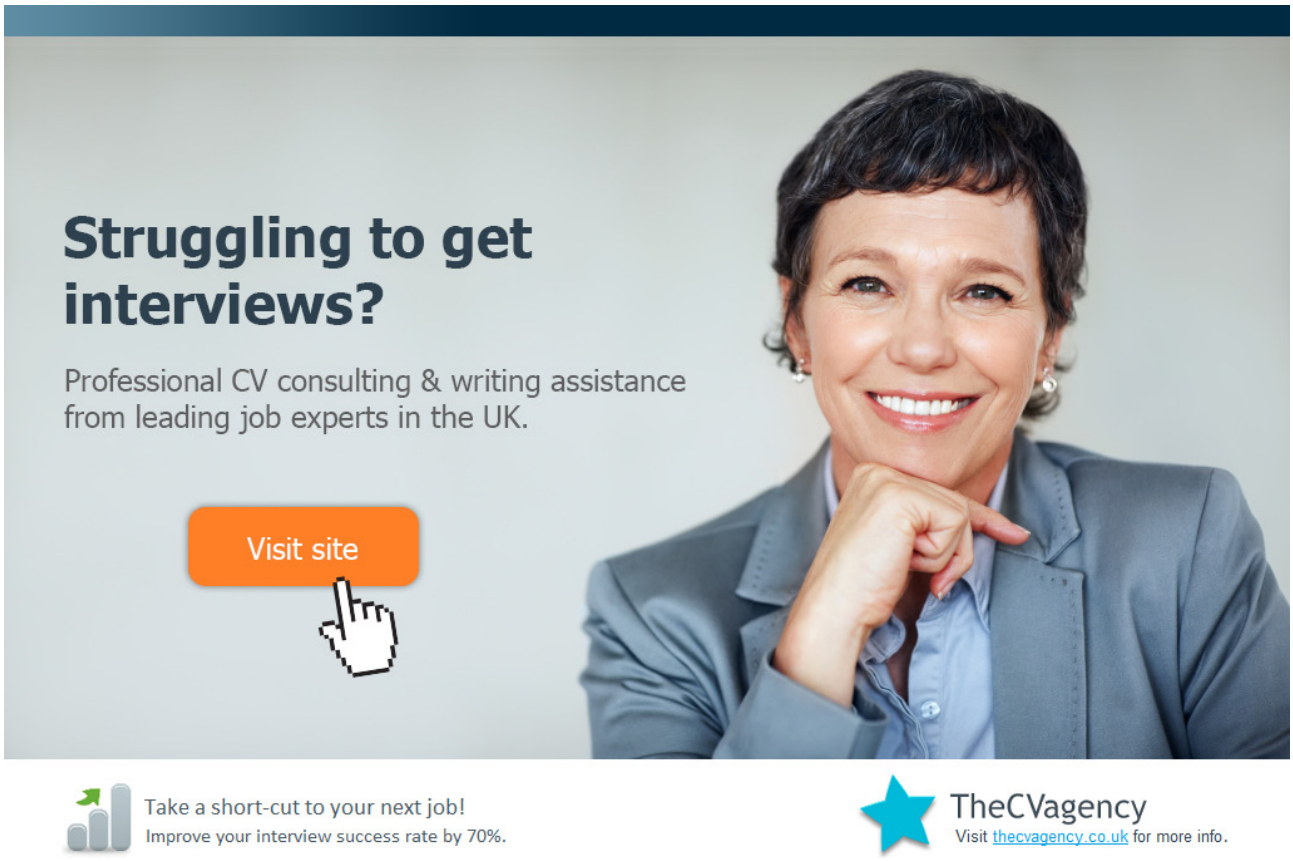

Download free eBooks at bookboon.com

Explanation

The method *PhoneOk()* checks whether a string is a legitimate phone number: It must consist of 8 digits. The method skipp any spaces (so it is permitted to enter spaces in a phone number), but since I do not want these spaces saved in the file the parameter to *PhoneOk()* is a reference parameter, so the spaces are removed.

The method *PhoneNumber()* is the input method, which returns a string consisting of the telephone and sales amounts that are separated by semicolons – or an empty string if the user just press Enter. Actually it is a relatively complex method that calls the method *Amount()* to enter an amount.

The program writes to the file in *Main()* and you should note that it is exactly the same algorithm as in the previous example. Generally, the example is somewhat more complex than the previous example, but it caused only the entry of data, and it's not the file handling, which complicates the task.



Download free eBooks at bookboon.com

Exam55

Read a comma separated file

It is a continuation of the example above, but where the task instead is to read the comma-separated file created in Exam54, summing the results for each phone number and print it on the screen. Since the same phone number can occur several times, it is necessary to read the entire file before the results can be printed. The individual numbers must be stored somewhere, and since we do not know the number of telephone numbers the program use a *Dictionary* with the phone number as key, while the value is a *List* of amounts:

```
private static Dictionary<string, List<double>> table =
   new Dictionary<string, List<double>>();
```

Reading the file is done in the the same way as in the first example of text files, and the difference is only that the treatment of a line that is read in the file this time is more complex. The program is written as follows:

```
class Program
{
   private static Dictionary<string, List<double>> table =
      new Dictionary<string, List<double>>();

   static void Main(string[] args)
   {
      StreamReader reader = new StreamReader("F:\\Temp\\Salg.txt");
      string line = reader.ReadLine();
      while (line != null)
      {
         ParseLine(line);
         line = reader.ReadLine();
      }
      reader.Close();
      Print();
   }

   static void ParseLine(string line)
   {
      string[] elem = line.Split(';');
      try
      {
         if (elem.Length == 0)
            Console.WriteLine("Empty line...");
         else
         {
            string phone = elem[0];
            List<double> list;
            if (table.ContainsKey(phone))
               list = table[phone];
         else
         {
            list = new List<double>();
            table.Add(phone, list);
         }
         for (int i = 1; i < elem.Length; ++i)
```


Download free eBooks at bookboon.com

```
        {
           try
           {
              list.Add(Convert.ToDouble(elem[i]));
           }
           catch
           {
              Console.WriteLine("Illegal ammount:" + line);
           }
        }
     }
  }
  catch
  {
        Console.WriteLine("Error: " + line);
     }
  }

  static void Print()
  {
     foreach (string phone in table.Keys)
     {
        Console.WriteLine(phone);
        List<double> list = table[phone];
        double total = 0;
        foreach (double number in list)
        {
           total += number;
           Console.WriteLine("{0, 12:F2} ", number);
        }
        Console.WriteLine("Ialt: {0, 12:F2}", total);
     }
  }
}
```

Explanation

The treatment takes place in the method *ParseLine()*. It splits a line at semicolons. The rest consists merely to test whether a phone number is already in the table, or whether it is a new number to be added to the table. Then the method parses the amounts. Note that they must be converted, and if there is an error an error message is written on the screen.

The last method *Print()* is a simple traversal of the input data, which writes the result on the screen.

Download free eBooks at bookboon.com

29 Binary files

Just as there are classes for working with text files, there are classes for the treatment of binary files, that is files that contains numbers or other binary data.

Exam56

Print 100 numbers in a fil

Write a program that writes 100 random numbers of the type *double* in a file named *Number.dat*.

How to

```
class Program
{
   static Random rand = new Random();

   static void Main(string[] args)
   {
      FileStream stream = new FileStream("F:\\Temp\\Numbers.dat",
      FileMode.OpenOrCreate, FileAccess.Write);
      BinaryWriter writer = new BinaryWriter(stream);
      for (int i = 0; i < 100; ++i) writer.Write(rand.NextDouble() * 10000);
      writer.Close();
      stream.Close();
   }
}
```

Click on the ad to read more

Download free eBooks at bookboon.com

Explanation

This time the file is represented by a *FileStream*. The constructor in addition to the file name has two additional parameters. The file *mode* indicates how the file should be opened. Here you specify that it must be opened if it exists, or else it is created. As an alternative, you can specify

- *CreateNew*, that creates a new file, but if it already exists, you get an exception
- *Create*, that creates a new file and if it already exists, it is overwritten
- *Open*, that opens a file and if it does not exist, you get an exception
- *Truncate*, that opens a file and empty it
- *Append*, that opens a file and places the file pointer at the end of the file – if the file does not exist, it is created

The last parameter specifies the access to the file, and this opens the file for writing. Alternatively, the file can be opened for reading or for both reading and writing.

After the file has been opened the program creates a *BinaryWriter* that is used to write numbers to the file. Note the method *Write()*, which has 18 overrides and then an override to each of the built-in data types.

Exam57

Read a binary file

Write a program that reads the content of the file generated by the above program and print the number of numbers that is read from til file, the sum of the numbers, the average and the smallest and largest number.

How to

```
class Program
{
   static void Main(string[] args)
   {
      double sum = 0;
      double min = double.MaxValue;
      double max = double.MinValue;
      int count = 0;
      FileStream stream =
         new FileStream("F:\\Temp\\Numbers.dat", FileMode.Open, FileAccess.Read);
      BinaryReader reader = new BinaryReader(stream);
      try
      {
         while (true)
         {
            double tal = reader.ReadDouble();
            sum += tal;
            ++count;
            if (min > tal) min = tal;
            if (max < tal) max = tal;
         }
```


Download free eBooks at bookboon.com

```
      }
      catch (EndOfStreamException)
      {
      }
      reader.Close();
      stream.Close();
      if (count > 0) Result(sum, min, max, count);
   }

   private static void Result(double sum, double min, double max, int count)
   {
      Console.WriteLine("It has been read {0} numbers", count);
      Console.WriteLine("The sum of the numbers are {0}", sum);
      Console.WriteLine("The average is {0}", sum / count);
      Console.WriteLine("The minimum number is {0}", min);
      Console.WriteLine("The maximum number is {0}", max);
   }
}
```

Explanation

Basically there is not much new, but there are two things to be aware of. One is how to handle end-of-file, and therefore when there are not more numbers in the file. To read the file you use a *BinaryReader*, and if you try to read beyond the end of the file, you get an exception. The other thing to consider is how to read the file with the method *ReadDouble()*.

This method reads the next 8 bytes and converts them into a *double* regardless of the content. It is the responsibility of the user (the user must know) that these 8 bytes actually represent a *double*. This information is not stored in the file. The *BinaryReader* class has similar methods to read other built-in types.

Exam58

Seek

A *FileStream* is always a sequence of bytes, and it is up to the program that reads the file, to interpret the content correctly. To a *FileStream* is associated a so-called file pointer, which is simply an integer that indicates the position in the file where the next read or write operation takes place. The program can manipulate the file pointer with a method *Seek()*, which can set the position of the file pointer anywhere – again it is up to the program to ensure that it is a meaningful position. The next example shows how to move the file pointer.

How to

```
public static void Main()
{
   FileStream stream = new FileStream("Tal.dat", FileMode.Create,
   FileAccess.ReadWrite);
   BinaryWriter writer = new BinaryWriter(stream);
   BinaryReader reader = new BinaryReader(stream);
```


Download free eBooks at bookboon.com

```
   for (int i = 0; i < 10; ++i) writer.Write(Math.Sqrt(i));
   stream.Seek(5 * sizeof(double), SeekOrigin.Begin);
   writer.Write(Math.PI);
   Console.WriteLine(reader.ReadDouble());
   stream.Seek(5 * sizeof(double), SeekOrigin.End);
   writer.Write(Math.E);
   stream.Seek(0, SeekOrigin.Begin);
   try
   {
      while (true) Console.WriteLine(reader.ReadDouble());
   }
   catch (EndOfStreamException)
   {
   }
   reader.Close();
   writer.Close();
   stream.Close();
}
```

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

Explanation

First you open a *FileStream*, but this time with readwrite access. Note also that the *mode* is *Create*, so that the file is created each time the method is performed. As a next step the stream are associated with both a writer and a reader, and then the program writes 10 numbers in the file. Then the program writes the number *pi* in the file, but first it positions file pointer, so the number is written as the 5th number (the first number is the 0th number). Since a *double* fills 8 bytes, the value must start with the byte number 40. After the number *pi* is written the program read a number, but when writing is moving the file pointer 8 bytes, it will be the next number that is read. Now move the file pointer 40 bytes, but this time from the end of the file. This means that the file pointer is positioned 40 bytes after the end of the file and the program writes the number *e*. The question is what happens to the intermediate places, which in principle are empty. The answer is that they are set to 0 – all bytes are 0. Finally the entire content of the file are read. Notice how the file pointer is first moved to the beginning of the file.

If the method is executed, the result is as follows:

Download free eBooks at bookboon.com

30 Info about directories and files

System.IO contains two classes that are useful to determine information about files and directories:

- *FileInfo*
- *DirectoryInfo*

The first represents a file, while the other represents a library or directory. There is not much to say about them, but they both have a number of properties that specifies information about a specific file or directory.

Exam59

FileInfo

Below is a program that prints information about a file:

```
static void Main(string[] args)
{
   string filename = "E:\\USA\\001.jpg";
   FileInfo info = new FileInfo(filename);
   Console.WriteLine(info.FullName);
   Console.WriteLine(info.Name);
   Console.WriteLine(info.CreationTime);
   Console.WriteLine(info.Attributes.ToString());
   Console.WriteLine(info.Length);
   Console.WriteLine(info.Extension);
   Console.WriteLine(info.DirectoryName);
}
```

Explanation

There is not much to explain, but the program requires that the file exists.

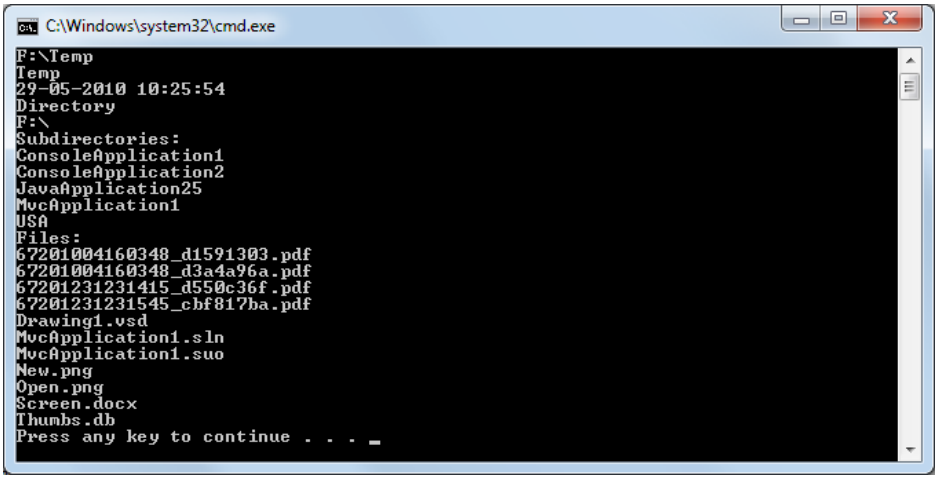
Exam60

DirectroryInfo

The next example is in principle identical to the foregoing but it print the information about a directory:

Download free eBooks at bookboon.com

```
static void Main(string[] args)
{
   string dirname = "F:\\Temp";
   DirectoryInfo info = new DirectoryInfo(dirname);
   Console.WriteLine(info.FullName);
   Console.WriteLine(info.Name);
   Console.WriteLine(info.CreationTime);
   Console.WriteLine(info.Attributes.ToString());
   Console.WriteLine(info.Root);
   Console.WriteLine("Subdirectories:");
   foreach (DirectoryInfo dir in info.GetDirectories()) Console.WriteLine(dir.Name);
   Console.WriteLine("Files:");
   foreach (FileInfo file in info.GetFiles()) Console.WriteLine(file.Name);
}
```



Click on the ad to read more
Download free eBooks at bookboon.com

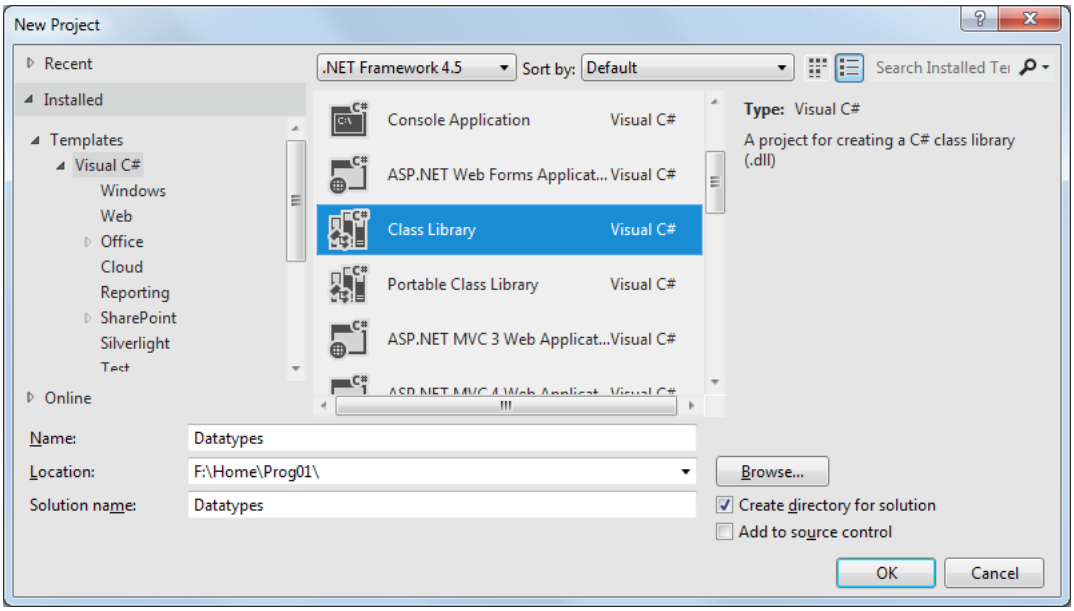
31 Object serialization

Object serialization is the process by which the state of an object is converted into a sequence of bytes, and thus as data that can be send to a stream, such as a file or over a communication line. It is possible to serialize arbitrary objects to a stream, and for that we can again read a serialized object, it is necessary together with the serialization of the object's data also to store information about the type. When you read a serialized object, it is call to deserialize the object.

Seen from the programmer is serialization and deserialization of objects simple and one should not write much, but if you think a little about it, it's a relatively complex process. An object is defined on the basis of a class and can contain simple variables, arrays and references to other objects, and not only that, so the class can also inherit variables from a base class. When an object must be serializable, all these elements are saved with all the information needed to build the whole structure again when the object is deserialized. The conclusion is that although the following examples show that it is simple to serialize objects, there are going on a lot behind, which is handled by the classes that underlie serialiseringen.

Datatypes

Before I show some examples of serialization I will create a class library. Open Visual Studio and select *Class Library* as a project template:



Visual Studio will then create a dll with classes. In this case the name is *Datatypes.dll*. After the project is created, it contains a class called *Class1*. I typically delete this class.

Download free eBooks at bookboon.com

I then add a new class, which I have called *Person*:

```
using System;

namespace Datatypes
{
   [Serializable]
   public class Person
   {
      private string name;
      private string position;

      public Person()
      {
         name = "";
         position = "";
      }
      public Person(string name, string position)
      {
         this.name = name;
         this.position = position;
      }

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

      public string Position
      {
         get { return position; }
         set { position = value; }
      }

      public override string ToString()
      {
         return name + ", " + position;
      }
   }
}
```

Then the project is translated.

Explanation

The class is quite simple and there is not much to explain, but the class is defined *Serializable*:

```
[Serializable]
public class Person
{
```

wand it is a necessary prerequisite that an object of the type *Person* can be serialized.

Download free eBooks at bookboon.com

Comment

A class library is not a program, and thus it is not something you can immediately try out, but it is a dll that can be used in other applications. What you should do is explained in the next example. A class library can hold more stuff, but will typically contain user-defined classes. The goal is a dll with classes that you often need and which are completed, tested and found free of errors. Such a dll can then be made available to other programs, and its classes will be used the same way as all other classes from .NET framework.

Exam61

Binary serialization

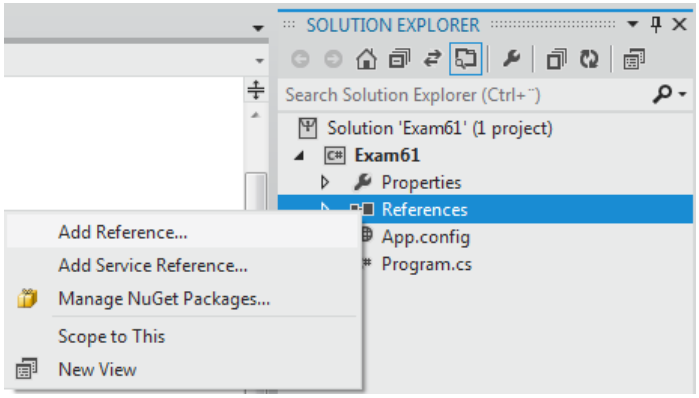
You should be writes a program that create a *Person* object and serialize it to a file.

How to

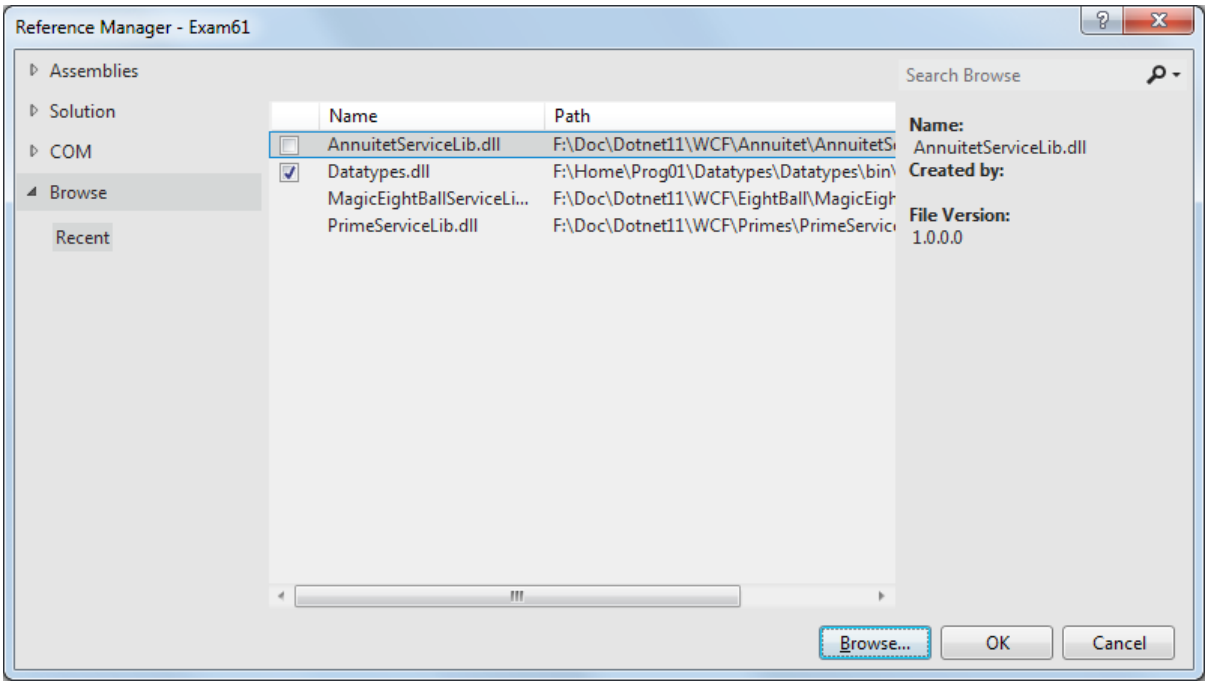
The start is as usual a *Console Application* project. This time I will start to associate the above dll to the project. In *Solution Explorer*, you right click on *References* and choose *Add Reference*…

This e-book
is made with
SetaPDF
SETASIGN
PDF components for PHP developers
www.setasign.com
Click on the ad to read more

Download free eBooks at bookboon.com

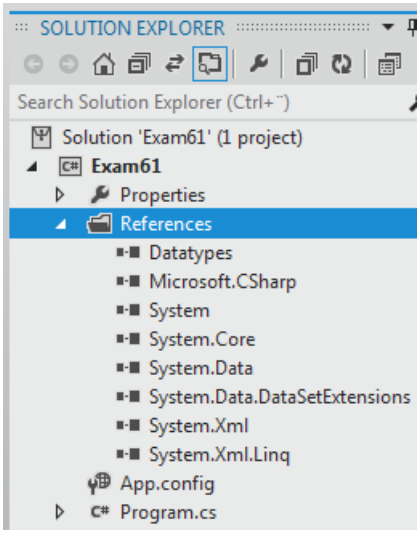


In the next window you click on the *Browse* button at the bottom of the window. You get a file browser and can browse through to the dll (*Datatypes.dll* in the project *Datatypes*):



When you then click *OK*, the dll is copied to the current project and its classes are now available for the program. If you then open the *References* in *Solution Explorer*, you can see which dll's the program has access to and it now include *Datatypes*. The other dll's are some that Visual Studio has automatically added references to, and they contain all the main classes in the .NET fra mework.

Download free eBooks at bookboon.com



After the dll with the class *Person* is attached to the program, the code is written in the following way:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using Datatypes;

namespace Exam61
{
   class Program
   {
      static void Main(string[] args)
      {
         Person pers = new Person("Valborg Kristensen", "Klog kone");
         Serialiser(pers);
      }

      private static void Serialiser(Person pers)
      {
         BinaryFormatter bf = new BinaryFormatter();
         FileStream stream = File.Create("F:\\Temp\\Person.dat");
         bf.Serialize(stream, pers, null);
         stream.Close();
      }
   }
}
```

Explanation

Note first the program's using statements. In addition to *System.IO* there is also defined a namespace

```
using System.Runtime.Serialization.Formatters.Binary;
```

which contains the classes that are necessary to be able to serialize an object. Note also that there is a definition of the namespace with the *Person* class:

Download free eBooks at bookboon.com

```
using Datatypes;
```

I *Main()* there is an object

```
Person pers = new Person("Valborg Kristensen", "Klog kone");
```

and the task is to write this object to a file. This means that it is necessary to store the two member variables *name* and *position* and the are themselves objects of the type *string*. One thing is to save the object, but then you also must be able to read the object again, and as is mentioned above you need also store information about the object's type. The object is stored as raw bytes, and for it again can be loaded as a *Person* object – consisting of two objects of the type *string* – should this information be saved with the object's data bytes. It is this process that is called serialization, and for this to be possible, both the object's type must be defined *Serializable* and, secondly, all the object's instance varaibler must be *Serializable*. In this case, all are met, as the *Person* class is defined *Serializable*, and the member variables has the type *string*, which is *Serializable*. The same is the case for all the simple types and many of the types .NET framework.

To serialize an object you must use a so-called *Formatter*. There are several options, and the different *Formatter* types are defined in the namespace *System.Runtime.Formatters*, but common to them is that they serialize an object with the information necessary for the object can be deserialized. They differ, however, regarding the format in which the object is stored in. In this case I use a binary formatter that store an object in a binary format.

Click on the ad to read more

Download free eBooks at bookboon.com

The method *Serialiser()* serialize an object of the type *Person* to the file *F:\Temp\Person.dat*. First you create a *BinaryFormatter* and then a file called *F:\Temp\Person.dat*. Finally, the formatter is used to serialize the object. It is thus relatively simple to serialize an object. One should be aware that when writing to a file, the code should usually be placed in a try block and a possible exception should be handled.

Exam62

Binary deserialization

The purpose of this example is to write a program that reads the object that is stored in the previous example and then print the object on the screen.

How to

First, is added in the same manner as above, a reference to the dll with the class *Person*. Then the program can be written as:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using Datatypes;

namespace Exam62
{
   class Program
   {
      static void Main(string[] args)
      {
         Person pers = DeSerialiser();
         if (pers != null) Console.WriteLine(pers);
      }

      private static Person DeSerialiser()
      {
         Person pers = null;
         try
         {
            BinaryFormatter bf = new BinaryFormatter();
            FileStream stream = File.OpenRead("F:\\Temp\\Person.dat");
            pers = (Person)bf.Deserialize(stream);
            stream.Close();
         }
         catch
         {
         }
         return pers;
      }
   }
}
```


Download free eBooks at bookboon.com

Explanation

The method *DeSerialiser()* tries to deserialize an object of the type *Person* saved in the file *F:\Temp\Person.dat*. First the program creates a *BinaryFormatter*. Next, it creates a stream to the file and the object is deserialized. The method returns the object if it is possible to deserialize a *Person* object (if the file exists and if it contains a *Person* object). If not, the method returns *null*. Note that it is necessary with a type cast to a *Person* object.

Comment

When you deserialize an object, it may be as the same type as the type the object is serialized as. Here we must remember that an object's type is also identified by the namespace that contains its type. That is why I in the above examples have placed the class *Person* in a class library so that *Person* is the same type in both programs.

Exam63

XML serialization

This example is, in principle, exactly the same as the previous two, but instead of using a *BinaryFormatter* the program use a *XmlSerializer*, so that the object is stored as XML, instead of binary data.

How to

Again, start with putting a reference to the dll with the class *Person*. The code is as follows:

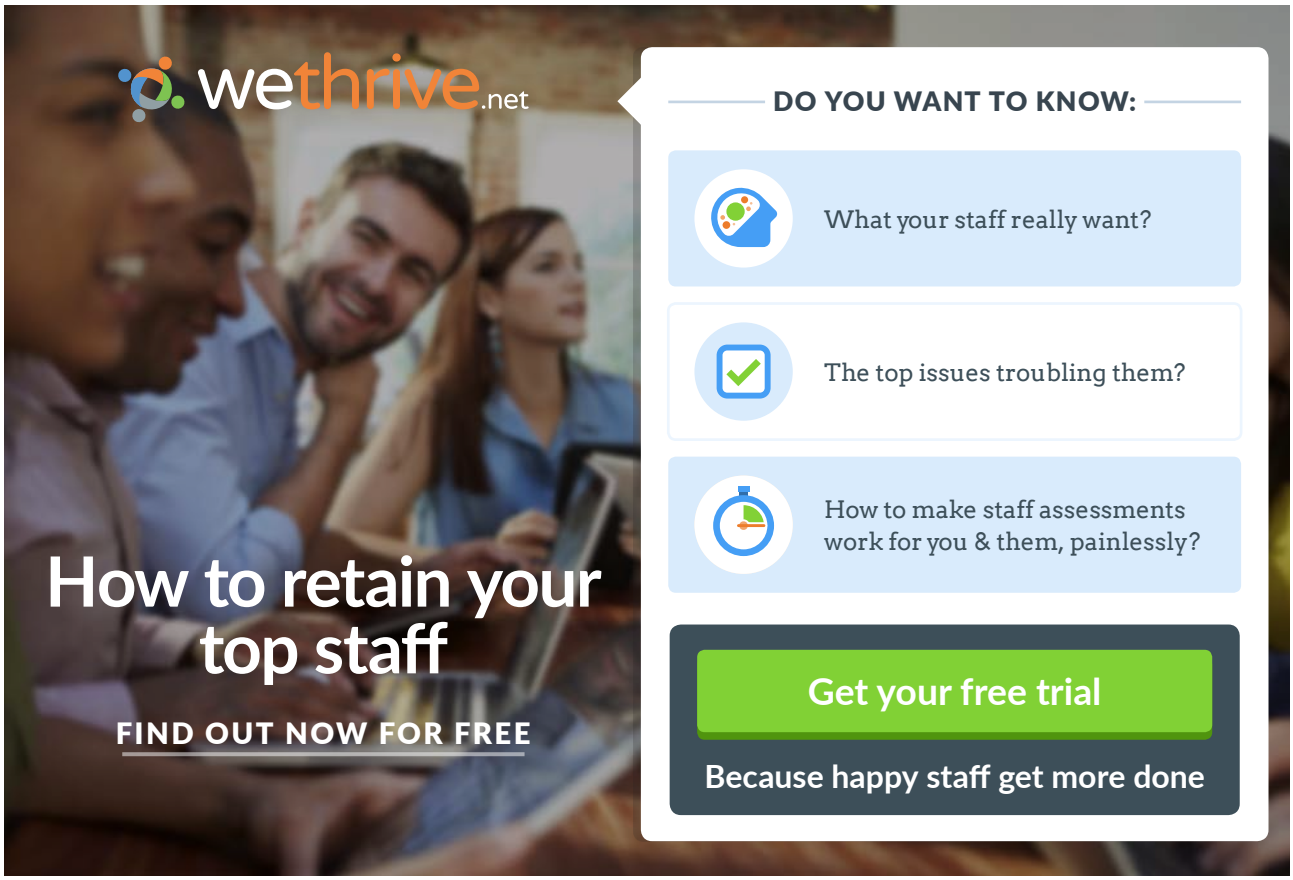
```
using System;
using System.IO;
using System.Xml.Serialization;
using Datatypes;

namespace Exam63
{
   class Program
   {
      static void Main(string[] args)
      {
         Save();
         Load();
      }
      public static void Save()
      {
         Person pers = new Person("Knud Andersen", "Skarpretter");
         Serialiser(pers);
      }

      public static void Load()
      {
         Person pers = DeSerialiser();
         if (pers != null) Console.WriteLine(pers);
      }
```


Download free eBooks at bookboon.com

```
      private static void Serialiser(Person pers)
      {
         XmlSerializer xs = new XmlSerializer(typeof(Person));
         FileStream stream = File.Create("F:\\Temp\\Person.xml");
         xs.Serialize(stream, pers);
         stream.Close();
      }

      private static Person DeSerialiser()
      {
         Person pers = null;
         try
         {
            XmlSerializer xs = new XmlSerializer(typeof(Person));
            FileStream stream = File.OpenRead("F:\\Temp\\Person.xml");
            pers = (Person)xs.Deserialize(stream);
            stream.Close();
         }
         catch
         {
         }
         return pers;
      }
   }
}
```

Explanation

The code is realy unchanged and the main difference is that the program uses a different namespace and thus also other serialization objects.



Click on the ad to read more

Download free eBooks at bookboon.com

The advantage to serialize an object as xml is that everyone who can read a text file can read and interpret it. It does not require knowledge of the binary format which is used of a *BinaryFormatter*.

The content of the file *Person.xml* are as follows:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <Name>Knud Andersen</Name>
     <Position>Skarpretter</Position>
</Person>
```

If you look at the xml that is generated, you can almost see that it is not enough that you can deserialize the object. For an object can be xml serializable's a there are few more things that must be met:

- there must be a default constructor
- there must be *get* and *set* properties for all instance variables

and both are satisfied of the class *Person*. Although binary serialization (and SOAP serialization as shown below) does not require it, it's a good idea to attach both a default constructor and the necessary properties to classes that are defined serializable.

Exam64

SOAP serialization

This example is exactly identical to the above example with the difference that this time the object is serialized by the SOAP protocol.

How to

As in other examples there must be a reference to the dll with the type *Person*, but this time there is a little extra observation. The class to soap formatting is in the namespace:

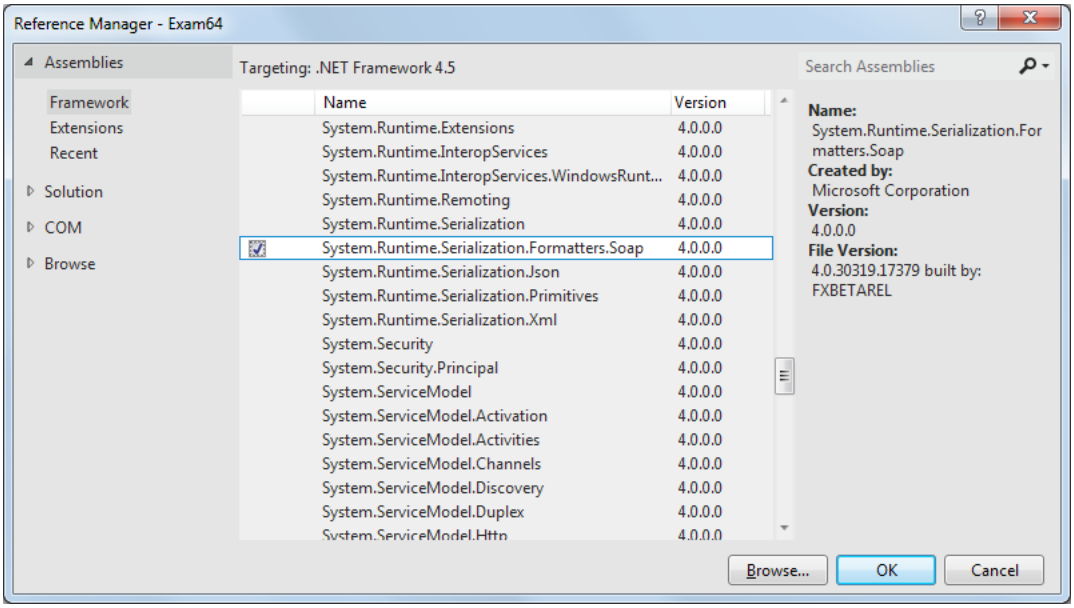
```
using System.Runtime.Serialization.Formatters.Soap;
```

but this namespace is not included in the dlls that Visual Studio makes references to as default. It is therefore necessary to set a reference to the namespace manually (see below). Then the code can be written:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
using Datatypes;

namespace Exam64
{
   class Program
   {
      static void Main(string[] args)
```


Download free eBooks at bookboon.com

```
        {
            Save();
            Load();
        }

        public static void Save()
        {
            Person pers = new Person("Gudrun Mortensen", "Heks");
            Serialiser(pers);
        }

        public static void Load()
        {
            Person pers = DeSerialiser();
            if (pers != null) Console.WriteLine(pers);
        }

        private static void Serialiser(Person pers)
        {
            SoapFormatter sf = new SoapFormatter();
            FileStream stream = File.Create("F:\\Temp\\Person.txt");
            sf.Serialize(stream, pers);
            stream.Close();
        }

        private static Person DeSerialiser()
        {
            Person pers = null;
            try
            {
                SoapFormatter sf = new SoapFormatter();
                FileStream stream = File.OpenRead("F:\\Temp\\Person.txt");
                pers = (Person)sf.Deserialize(stream);
                stream.Close();
            }
            catch
            {
            }
            return pers;
        }
    }
}
```



Download free eBooks at bookboon.com

Explanation

I will not interpret the result (ie explain the SOAP protocol), but it is important to note that it is XML, and thus is text and thus in principle that everyone can read the content.

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <a1:Person id="ref-1" xmlns:a1=
            "http://schemas.microsoft.com/clr/nsassem/Datatypes/
            Datatypes%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%
            20PublicKeyToken%3Dnull">
         <name id="ref-3">Gudrun Mortensen</name>
         <position id="ref-4">Heks</position>
      </a1:Person>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

Download free eBooks at bookboon.com

Exam65

Serialization of a collection

The aim of this example is to show that you can also serializes a collection of objects, and in this case a collection of *Person* objects.

```
namespace Exam65
{
   class Program
   {
      static void Main(string[] args)
      {
         Save();
         Load();
      }

      public static void Save()
      {
         List<Person> list = CreateList();
         Serialiser(list);
      }

      public static void Load()
      {
         List<Person> list = DeSerialiser();
         if (list != null) Show(list);
      }

      private static List<Person> CreateList()
      {
         List<Person> list = new List<Person>();
         list.Add(new Person("Frede Andersen", "Mestersvend"));
         list.Add(new Person("Gudrun Knudsen", "Spåkone"));
         list.Add(new Person("Karl Petersen", "Natmand"));
         list.Add(new Person("Maren Hansen", "Sin mands kone"));
         return list;
      }

      private static void Show(List<Person> list)
      {
         for (int i = 0; i < list.Count; ++i) Console.WriteLine(list[i]);
      }

      private static void Serialiser(List<Person> list)
      {
         BinaryFormatter bf = new BinaryFormatter();
         FileStream stream = File.Create("F:\\Temp\\List.dat");
         bf.Serialize(stream, list);
         stream.Close();
      }

      private static List<Person> DeSerialiser()
      {
         List<Person> list = null;
         try
         {
            BinaryFormatter bf = new BinaryFormatter();
            FileStream stream = File.OpenRead("F:\\Temp\\List.dat");
            list = (List<Person>)bf.Deserialize(stream);
            stream.Close();
```


Download free eBooks at bookboon.com

```
            }
            catch
            {
            }
            return list;
        }
    }
}
```

Explanation

There is really not much to explain, and the only thing to point out is that the various collection classes generally are *Serializable*, and a list of *Serializable* objects (objects of the type *Person*) can therefore immediately be serialized.

Download free eBooks at bookboon.com

32 User defined serialization

The serialization process as described is generally seen by the programmer simple and there is rarely reason to do more than are described above. However, there is the possibility to intervene in the process if there are special needs.

As an example one can imagine that a class has variables that you, for one reason or another, do not want to be serialized. Below is a class *Employee* which inherits the class *Person* and expands it with two variables, the first being the date of employment, while the second is the current year. The class has a method *Seniority()*, which – somewhat simplified – returns an employee's seniority as the number of years between year of employment and the current year. The class is *Serializable*, but it makes no sense to serialize the variable year, since its value depends on when the program is run. However, one can mark a variable with the attribute *NonSerialized*, which says that it should not be serialized.

```
[Serializable]
public class Employee : Person
{
   private DateTime date;

   [NonSerialized]
   protected int year = DateTime.Now.Year;
```

Click on the ad to read more

Download free eBooks at bookboon.com

```
   [OnDeserialized]
   public void InitYear(StreamingContext context)
   {
      year = DateTime.Now.Year;
   }

   public Employee(string name, string position, DateTime date)
      : base(name, position)
   {
      this.date = date;
   }

   public int Seniority()
   {
   return year - date.Year;
   }
}
```

It does provide a small problem with getting this variable initialized when an object is deserialized. This is handled by the method *InitYear()*, which is a method which is performed after the deserialization process. Here you must note the attribute, which tells that it is a method to be performed after deserialization and the method's signature and the parameter type *StreamingContext* that is an enum that is rarely used for anything.

The following program serializes and deserializes an *Employee* object:

```
namespace Exam66
{
   class Program
   {
      static void Main(string[] args)
      {
         Save();
         Load();
      }

      public static void Save()
      {
         Employee employee = new Employee("Karlo Andersen", "Tater",
                             new DateTime(2002, 03, 15));
         Serialiser(employee);
      }

      public static void Load()
      {
         Employee employee = DeSerialiser();
         if (employee != null) Console.WriteLine(employee);
      }

      private static void Serialiser(Employee employee)
      {
         BinaryFormatter bf = new BinaryFormatter();
         FileStream stream = File.Create("Employee.dat");
         bf.Serialize(stream, employee, null);
         stream.Close();
      }
```


Download free eBooks at bookboon.com

```
      private static Employee DeSerialiser()
      {
         Employee employee = null;
         try
         {
            BinaryFormatter bf = new BinaryFormatter();
            FileStream stream = File.OpenRead("Employee.dat");
            employee = (Employee)bf.Deserialize(stream);
            stream.Close();
         }
         catch
         {
         }
         return employee;
      }
   }
}
```

There are other corresponding attributes that can be used to decorate the methods of performing a similar signature:

- *OnDeserializing* to provide a method which is carried out before the deserialization process
- *OnSerialized* specifies a method that is performed immediately after the serialization process
- *OnSerializing* which specifies a method being carried out before the serialization process

In addition to that as illustrated above where you can define methods which are performed before and after, respectively serialization and deserialization, it is also possible to directly control the process in terms of how data is serialized and deserialized. The principle is illustrated by the following class that represents a member who is a specialization of the class *Person*:

```
[Serializable]
public class Member : Person, ISerializable
{
   private string ssn;
   private string phone;
   private string email;

   public Member(string ssn, string name, string position, string
          phone, string email)
      : base(name, position)
   {
      this.ssn = ssn;
      this.phone = phone;
      this.email = email;
   }

   public Member(SerializationInfo info, StreamingContext context)
   {
      ssn = info.GetString("ssn");
      Name = info.GetString("name");
      Position = info.GetString("position");
      phone = info.GetString("phone");
      email = info.GetString("email").ToLower();
   }
```


Download free eBooks at bookboon.com

```
public string Ssn
{
   get { return ssn; }
   set { ssn = value; }
}

public string Phone
{
   get { return phone; }
   set { phone = value; }
}

public string Email
{
   get { return email; }
   set { email = value; }
}

public void GetObjectData(SerializationInfo info, StreamingContext context)
{
   info.AddValue("ssn", FormatSsn());
   info.AddValue("name", Name);
   info.AddValue("position", Position);
   info.AddValue("phone", FormatPhone());
   info.AddValue("email", email.ToUpper());
}

private string Format Ssn()
{
   StringBuilder builder = new StringBuilder();
   for (int i = 0; i < ssn.Length; ++i)
      if (ssn[i] >= '0' && ssn[i] <= '9')
      {
```

Unlock your potential
eLibrary solutions from bookboon is the key
bookboon eLibrary
Interested in how we can help you?
email ban@bookboon.com
Click on the ad to read more

Download free eBooks at bookboon.com

```
            builder.Append(ssn[i]);
            if (builder.Length == 6) builder.Append('-');
         }
      return builder.ToString();
   }

   private string FormatPhone()
      {
         StringBuilder builder = new StringBuilder();
         for (int i = 0, j = 0; i < phone.Length; ++i)
         if (phone[i] >= '0' && phone[i] <= '9')
         {
            builder.Append(phone[i]);
            ++j;
            if (j == 2)
            {
               builder.Append(' ');
               j = 0;
            }
         }
      return builder.ToString();
   }

   public override string ToString()
   {
      return string.Format("{0}\n{1}\n{2}\n{3}", ssn, base.ToString(), phone, email);
   }
}
```

It is important to note that the class implements the interface *ISerializable*, which is an interface that defines a single method: *GetObjectData()*. If a serializable class implements this interface, a formatter will call the method *GetObjectData()*, and here you can so in the form of key / value pairs define how each element should be serialized. A key is a string, and often you will use the name of a variable, but it is not a requirement. In this case – just to do something – the method will ensure that a social security number is saved as 6 digits, a hyphen and 4 digits and a phone number is stored as pairs of digits separated by spaces, and that your email address only has capital letters. Note that this method does not check if the social security number and the telephone number are legitimate. In practice, it should be checked somewhere, but it's not here not to make the code more complex than necessary.

The class also has a new constructor there among other things has a *SerializationInfo* parameter. This constructor is needed to deserialize an *ISerializable* object, and here you can with a series of get methods gets the values which are serialized in the method *GetObjectData()*.

```
namespace Exam67
{
   class Program
   {
      static void Main(string[] args)
      {
         Save();
         Load();
      }
```


Download free eBooks at bookboon.com

```
   public static void Save()
   {
      Member member = new Member("1234567890", "Nikoline Frederiksen",
              "SinMandsKone", "87654321", "niko.Frede@mail.DK");
      Serialiser(member);
   }

   public static void Load()
   {
      Member member = DeSerialiser();
      if (member != null) Console.WriteLine(member);
   }

   private static void Serialiser(Member member)
   {
      BinaryFormatter bf = new BinaryFormatter();
      FileStream stream = File.Create("Member.dat");
      bf.Serialize(stream, member, null);
      stream.Close();
   }

   private static Member DeSerialiser()
   {
      Member member = null;
      try
      {
         BinaryFormatter bf = new BinaryFormatter();
         FileStream stream = File.OpenRead("Member.dat");
         member = (Member)bf.Deserialize(stream);
         stream.Close();
      }
      catch
      {
      }
      return member;
   }
}
}
```

Download free eBooks at bookboon.com

Part 5 Final examples

I will finish the book by showing two programs, which is slightly larger than the book's other examples, and the goal is to have a little more focus on the development process and to demonstrate the use of the book's many concepts in some larger context. The goal is therefore to focus on the use of the book's substance, and examples such as adding nothing new.

Lottery

This example has primarily focused on classes and the choice of classes and how these classes interact to solve the specific task.

The task

The task is to write a console application that can print lottery tickets, but also can be used from this week's lottery numbers to determine the number of correct rows. The program must be used as a command from a prompt, where you with using of options can specifie what to do.

A typical use of the program is that you use the program to form the rows that you want to play. After this week's winning numbers are drawn, you use that program to determine the game's outcome.

There are several kinds of lottery, and as an example you have the normal lottery with 36 numbers and the Wednesday lottery with 48 numbers. The program should be prepared in such a way that it can be applied to all of the lotto games, which is of that nature.

Comment

The above description does not describe the job adequately. There are several things that must be addressed before one can tackle the task and especially the user's interaction with the program and thus the format of the individual commands, like the need for a clarification of the format of the programs result.

The development of a program may start with such a clarification of the task, a job which is usually called for *analysis* and the result is a requirements specification, where the final requirements for the finished program are established. The work can be anything from simple clarification of issues related to the task to a phase that extends over time and where one might only specify parts of the task at a time. For very large programs, especially if there is uncertainty attaching to the solution, experience shows that it is impossible to specify all requirements in advance, but that we must continually work with the requirements as to get greater insight into and understanding of the task to be solved. The result is that the analysis is important and necessary, and in practice the analysis can be extensive and time consuming. On the other hand one must not underestimate the importance, since wrong decisions can be disastrous for the finished product.

Download free eBooks at bookboon.com

Analysis

In this task, I will define a lottery game as a number of lottery rows, and in which a lottery row consists of a number of different lotto numbers, which are integers within a range. A lotto game is characterized by the following parameters:

-a	the smallest allowable number	1
-b	the largest allowable number	36
-r	number of lotto numbers per row	7
-n	number of rows to be played	

where the last column is a default value. This means that I have chosen a normal (Danish) lottery as default. These values can be provided as options on the command line.

You can also set the following options:

-o	output file for the result, which as default is the screen
-i	input file with numbers of the week's lottery game that must be checked
-u	this week's lotto (winning) numbers separated by spaces

The two latter options are used to control the lotto numbers for the current week, while the other five options are relating to form the rows for a new lottery game.

Click on the ad to read more

Download free eBooks at bookboon.com

The program will be called *lotto*, and below are examples of legal commands:

```
lotto -n 20
lotto -a 1 -b 48 -r 6 -n 20 -o uge42.txt
lotto -i uge42.txt -u 2 13 18 22 34 35
lotto -i uge42.txt -u 2 13 18 22 34 35 -o uge42Res.txt
```

Generally there are the following requirements for the arguments on the command line:

- All options that have a default value (a, b, r, o) may be omitted.
- Options may appear in any order.
- There must be no conflicting options (eg. -r and -i) on the same command line.

The concept of a lottery coupon is not included in the program because it does not make sense – a lottery game is in principle one big coupon that can have an arbitrary number of rows.

There are following requirements to the lotto rows:

- The numbers in each lotto row must be sorted in ascending order.
- The same number must not occur more than once.
- The rows must each be sorted in ascending order (after the first number after the next number, etc.).
- The first row defines the game and contains the date, the smallest and largest legal lottery number and the number of lottery numbers per. row.

The result of the verification must be a file (the screen by default) that contains all rows, but for each row is added a number indicating the number of correct lottery numbers. The result should be sorted by this number in descending order.

A particular issue is error handling. If you enter an illegal command, for example because it is incorrect, missing or conflicting options, illegal lotto numbers, illegal files, etc., the program will terminate with an error message and the program's syntax. It is standard action for commands.

If in a check of the week's lottery there is an illegal row, the number of correct lottery numbers are replaced with a short error message, and possibly illegal rows must be placed last in the result.

One last thing concerning the program parameters:

$a \geq 1 \wedge a < 100$
$b \geq a + 5 \wedge b < 100$
$r > 2 \wedge r \leq 5$
$r < b - a$

Download free eBooks at bookboon.com

Stated slightly differently, a lottery number can't exceed two digits, and a lottery row can have up to 15 numbers. Similarly, there are proposed some minimum requirements. There are not really so many reasons for these requirements except that it makes it easier to format the result, and it avoids some special cases, such as that the game can be empty, which in practice is uninteresting. The requirements are based on practical games, so the program can play typically occurring lottery games.

Comment

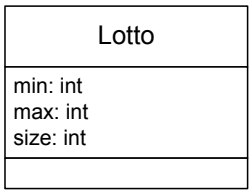
Above, I have in an analysis defined the requirements for the program and then you can start the work to write the program, but one should also start with some thoughts about how the program should be made including which classes the program will consist of. It is a work which in practice is called for design.

When making an application, it is important that the program is error free and meets the requirements. Is it the case, the task is in principle solved, but there are also other goals that are important. In particular, it is important that the program is designed in such a way that in future it is easy to maintain the program. All programs must be maintained and modified over time and therefore the program must be designed such that it is easy to read and understand, and it must be designed so that a change in one place does not mean that the change is spreading across the whole program, so very large parts of the code should be changed. That's why you should spend time on design, so you do the necessary considerations of program architecture. One can think of design as a place where you make working drawings, something you do in every other places where you have to build or produce anything.

In addition to maintenance performance can also be important, and it is a matter of algorithms. To write good algorithms is also a design activity.

Design

The centerpiece of the above program is a lottery row, which consists of lottery numbers corresponding to the current lottery game. A program must therefore be associated with an object that defines the game:



and the program must have exactly one of those objects which are available for the program's other classes.

Download free eBooks at bookboon.com

A lottery number is not much more than an integer with a random value within the range defined by the *Lotto* object

LottoNumber
rand: Random number: int

and an object of the type *LottoNumber* should not be able to change value. A *LottoRow* consists of a number of *LottoNumber* objects and an integer that indicates how many there are correct:

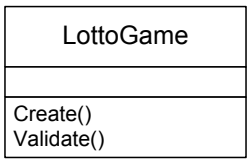
LottoRow
correct: int
Validate(LottoRow)

It is a relatively complex class. The method *Validate()* will validate the row from the weekly lottery and determine the number of right numbers. Moreover, the class should have a constructor, which aims to ensure that all *LottoNumber* objects are different, and the row is sorted in ascending order.

Click on the ad to read more

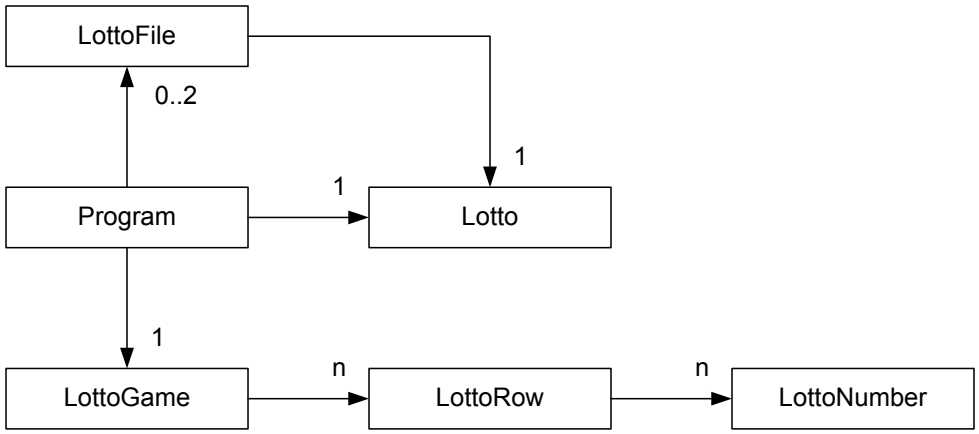
Download free eBooks at bookboon.com

The game itself is defined as a class that has a number of lottery rows, and is the program's most important class:



The class should have two methods, that respectively create lottery rows and validate an existing game up against this week's lottery numbers. Both methods are relatively complex and must include sorting of the lottery rows, but each on there own criteria. It is also these methods that must print the result to a file and read an existing game from a file.

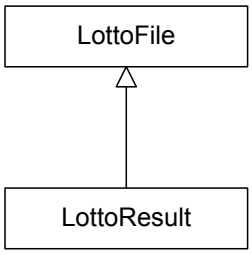
The relation between the above classes can be illustrated in the following figure:



Here the class *LottoFile* is a text file that contains lottery rows. Actually there are two kinds of files:

- one containing lottery rows for a new game – formed by the method *Create()* in *LottoGame*
- one containing lottery rows that are validated against this week's lottery numbers – formed by the method *Validate()* in *LottoGame*

One can therefore consider whether, instead of to use a design that is a specialization:



Download free eBooks at bookboon.com

Finally there is the class *Program*, which is the class with the *Main()* method. It is indeed a complex class, since it is that class that must validate the parameters from the command line.

With the above, it is the essentials regarding the programs classes and hence the program's architecture is in place.

Then there are the algorithms, and immediately the following algorithms are not trivial:

- Sorting the lottery rows and a collections / arrays of lottery rows
- Validating the command line in *Main()*
- The method *Create()* in *LottoGame*
- The method *Validate()* in *LottoGame*

Sorting is generally not straightforward and there are several algorithms, each with their advantages and disadvantages, but sorting is actually a complete method in the .NET framework, so there is no need to write your own sorting method.

Then there is the validation of the command line, which should be done as follows:

```
if option -i occurs then Check for Validate or Check for Create
if there is an error then
{
   print an error message
   terminate the program
}

Check for Create until there is an error or all arguments are ok
validate option -a, -b, -r
if ok then create a Lotto object
validate n, that must be greater than 0
if there is an -o option then validate the filename or let the name
be the screen
if ok then create a LottoFile object

Check for Validate until there is an error or all arguments are ok
validate input file and check that it is a LottoFile
if ok then create a Lotto object from the first line in the input file
Validate the week's lottery numbers
if ok then Create a LottoRow object for the week's lottery
if there is an -o option then validate the filename or let the name be the screen
if ok then create a LottoFile object
```

The algorithm to create lottery rows consists primarily of a loop that creates rows in accordance with the requirements and prints the finished result in a file:

Download free eBooks at bookboon.com

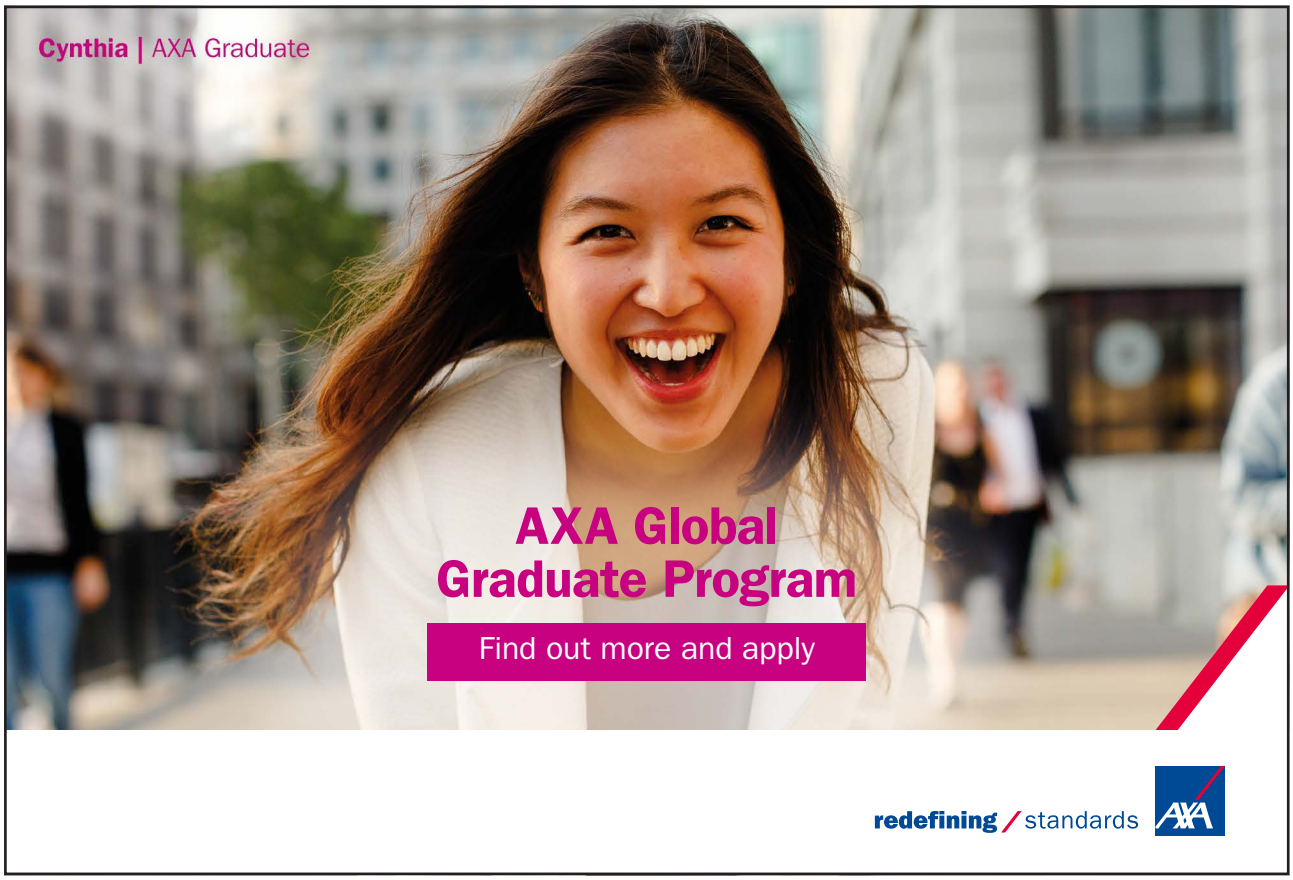
Metoden *Validate()*

```
Create a container (array, list) for the lottery rows
repeat until all rows are created
{
   create a new row
   if the row not already are in the container then add it to then container
}
Sort the container in increasing order
for each row in the container
{
   print the row to the output file
}
```

Finally you have from the weekly lottery numbers (winning numbers) to validate a game for the number of correct numbers and print the result in a file:

Metoden *Validate()*

```
Create a contain (array, list) to the lottery rows
repeat for each row i the input file
{
   if row is a legal row then
   {
      validate row against the weekly lottery
   }
   or
   {
      mark the row as illegal
   }
```



Click on the ad to read more

Download free eBooks at bookboon.com

```
   add the row to the container
}
Sort the container in descending order
for each row in the container
{
   print the row to the output file
}
```

Comment

After you have completed a design, you have an overall architecture for the program and have made important decisions for the finished program. You are now ready to write program's code, but no matter how careful you were during both the analysis and the design, it may happen that there will be new things, and perhaps you even have to create new requirements, and then the requirements specification must be updated. In the same way, the programming often lead to the occurrence of changes to the design, with possible new classes which influence the application architecture. Where applicable, the design must also be updated.

There are essentially two reasons to make a design. The key is that the design work is a necessary process to ensure that important decisions are in place before tackling the programming process and the other is that the result of a design can later serve as documentation of how the application is made. For the sake of the final reason the design should be updated with significant changes as the documentation value would not exist.

Regarding the first with the design as an important activity towards the final program, you should be aware of the level of detail. Design is referred to as a process whose purpose is to make important decisions about the program architecture and make choices regarding algorithms etc. The design must not be too detailed, and there's nothing wrong with that there in the programming are created classes that do not appear in the original design. It is typical classes that do not represent the key concepts within the program area of concern, but are classes as a programmer establish, because they are useful to get a good, maintainable code out of it. Conversely, there may also happen that programming has created new classes of important concepts, and is it the case, the design must be updated. Regarding the level of detail you should always keep in mind that a design must provide an oversight and establish the basic architecture, but a design must not be so detailed that it begins to resemble the finished program, written using diagrams and pseudo code. If so, you have created something that still does not fit on the finished result, and at best you have not been achieved other than to spend your time in an inappropriate manner.

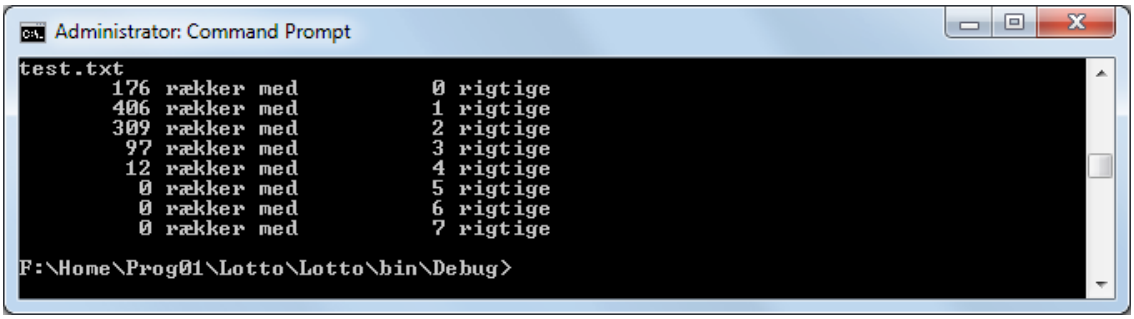
Programming

I will not show the code here because it is relatively extensive, but I will refer to the finished code, which, incidentally, is fully documented with comments. I will instead focus on what's happened with the requirements and design after the code is finished.

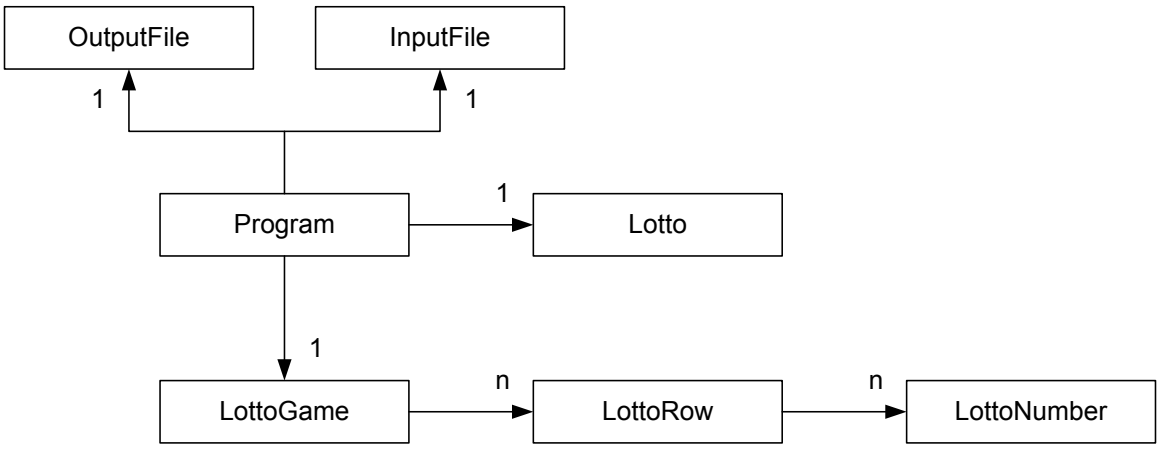
Download free eBooks at bookboon.com

Requirements are extended with a mention that the game should not take into account the concept supplementary lottery numbers.

Moreover, there is a single addition. Once you've created a lottery game and want to validate it against this week's lottery, the result should be followed by a distribution that shows how many rows there are with 1 right how many rows there are 2 right, etc. The result could, for example be the following:



In the design I have only made one important change. Originally, I worked with a *LottoFile* and possible a specialization of it. I have changed the design so that there instead are two *File* types. Its overall architecture is therefore:



You can get Visual Studio to create a class diagram. In Solution Explorer, you choose

Add | New Item

and here you choose *Class Diagram*. You can then drag classes from *Solution Explorer* into the chart and get a picture that resembles the following (see next page). One should not overestimate the importance of this facility, but the chart can be used as a documentation of how the program is built.

Download free eBooks at bookboon.com

Test

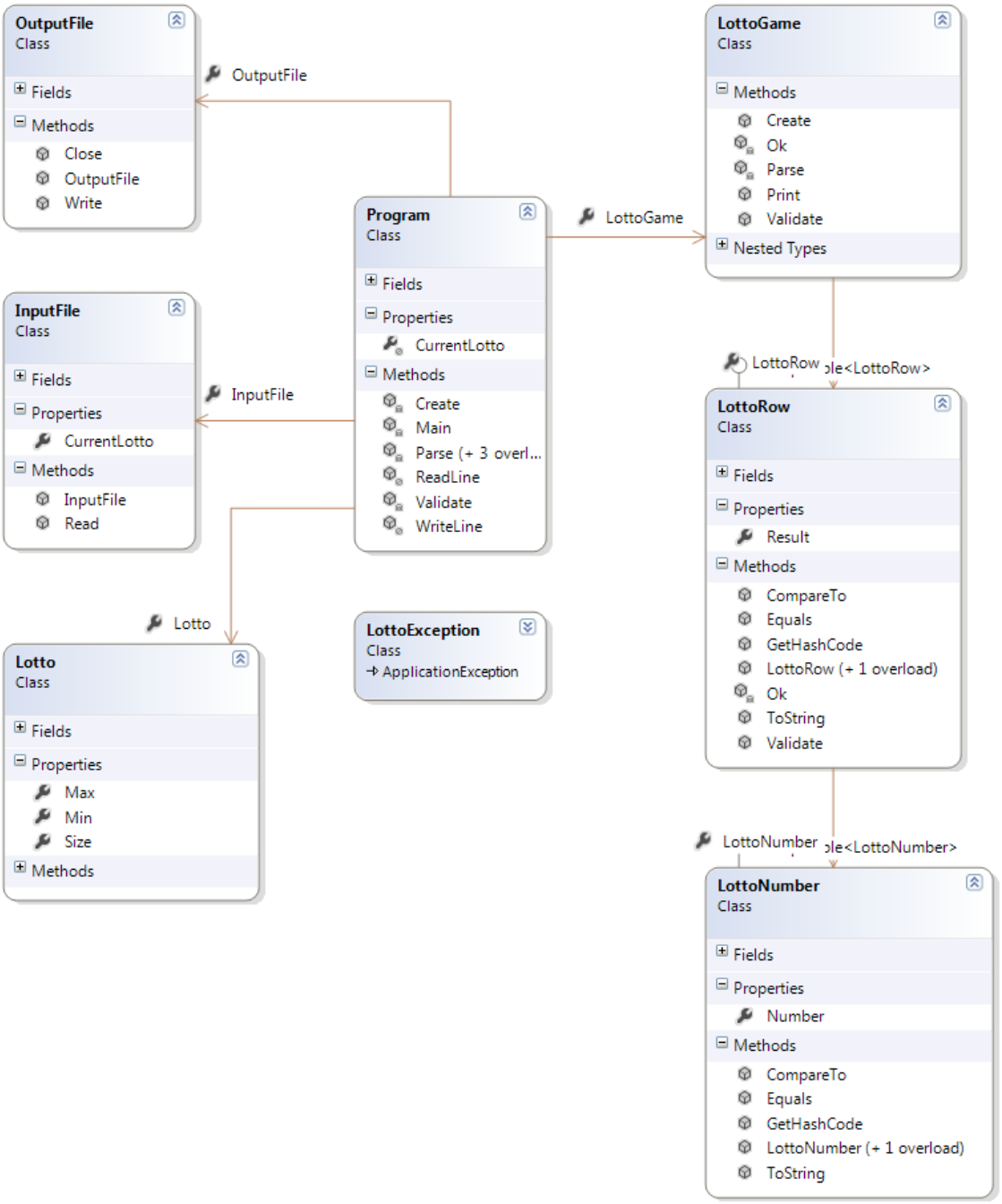
When the program is written, it must be tested, and possibly errors must be corrected. It sounds simple, but it's not, and if not you do it systematically, you will almost certainly come to overlook some situations and have the risk that the program comes with errors. Even this program, which after all is not very large, requires that you must do some considerations concerning tests.

I will split the test in three parts:

- That options are handled properly
- That the result is correct
- How the program behaves at a large number of rows

Click on the ad to read more

Download free eBooks at bookboon.com



For testing, be sure to test both for a legitimate input and an illegal input, and in all cases, check whether you get the expected result.

Download free eBooks at bookboon.com

In this case it is the possibility for 7 different options, which can either be there or not be there and if one adds an extra possibility as an illegal option there are 8 options. For each of these options, there are three possibilities corresponding to that which may be a legal value, an illegal value or the value may be missing, and therefore there are somewhat similar to $3 \cdot 2^8 = 768$ possibilities for arguments on the command line. Since you also must test whether the sequence of each option means something, you can see that there will be an exceedingly large number of cases to test – so great that in practice it is impossible to verify all cases by simple testing. This should be compared with that when a test fails, the error must be corrected, and all tests should in principle go on again to be sure that the changes you have made does not have unintended side effects.

In such a situation, sometimes I write a bat file with commands:

```
lotto
lotto -n 5
lotto -n 5 -a 1 -b 48 -r 6
lotto -n 5 -a 1 -b 48 -r 16
lotto -n 5 -a 1 -b 48 -x 5 -r 6
lotto -n 5 -a 1 48 -b -r 6
lotto -n 20 -o test1.txt
lotto -o result1.txt -i test1.txt
lotto -o result1.txt -i test1.txt -u 2 5 10 15 20 25 30 35
lotto -o result1.txt -i test1.txt -u 5 10 15 20 25 30 35
lotto -o result1.txt -u 5 10 15 20 25 30 35 -i test1.txt
lotto -b 48 -r 6 -n 10000 -o test2.txt
lotto -i test2.txt -o result2.txt -u 8 16 24 32 40 48
```

Here are some commands are legitimate, others are illegal. If the file is called *Test.bat* and you then run the commands like:

```
Test > test.txt
```

you get a file with the results, where it is easy to verify that the commands are executed correctly, and above all it is easy to repeat the testing.

Expression

The program *Lotto* has primarily focused on classes and choice of classes, while the next program has greater focus on algorithms.

The task

The task is to write a program where you can enter a mathematical expression that depends on one or more variables, for example:

$$2 * \sin(x0) + sqrt(5 * x1 + 3)$$

Download free eBooks at bookboon.com

which depends on two variables. Then, you can enter values for the expression's variables and the program must then evaluate the expression of these values.

If there is an error:

- you enter an expression that is not syntactically correct
- you enter arguments that are not legal for the current expression
- an error occurs when the expression is evaluated

the program must print an error message.

In the example above, the expression includes sinus and square root. The program must support the most common mathematical functions somewhat similar to what is possible with a mathematical calculator, but it is a desire that it should be easy to expand the program with new functions, if the need arises.

In this case, the program is a console application, but it's a wish that the basal parts of the code concerning expressions can be used in other applications and, therefore, as well as possible is separated from the part which has to do with the user's input.

Click on the ad to read more

Download free eBooks at bookboon.com

Analysis

Before you can solve the task there are a few things to be clarified:

- what is an expression exactly – what is the syntax for an expression
- what is meant by evaluating an expression
- how should the user interaction to be

Regarding the first, the requirements are the following:

- An expression is not case sensitive and it should not matter if you write it with small or big letters.
- An expression may contain any number of variables, referred to as *x0*, *x1*, *x2*, …, that is an *x* followed by a non-negative integer. There should be no requirement that the numbering is consecutive. Finally, it should be allowed simply to write *x* as an alias for *x0*.
- You should always indicate numbers (constants) with a period as decimal point.
- An expression must support the four general operators +, -, * and /.
- It should be allowed to use parentheses.
- If a mathematical function has multiple arguments, they are separated by commas.

A mathematical function is identified by a name, and after the name there may be a parameter list given in parentheses. In an expression, you can use the following functions:

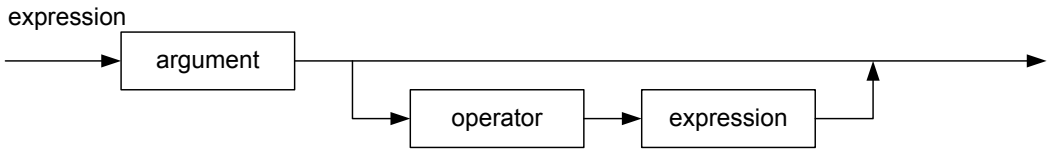
- constant functions (functions without parameters)
  - pi
  - e
- functions in 1 variable
  - sin
  - asin
  - cos
  - acos
  - tan
  - atan
  - cot
  - acot
  - ln
  - exp
  - log
  - alog
  - sqr

Download free eBooks at bookboon.com

    - sqrt
    - abs
    - frac
    - floor
- functions in 2 variables
    - pow
    - root

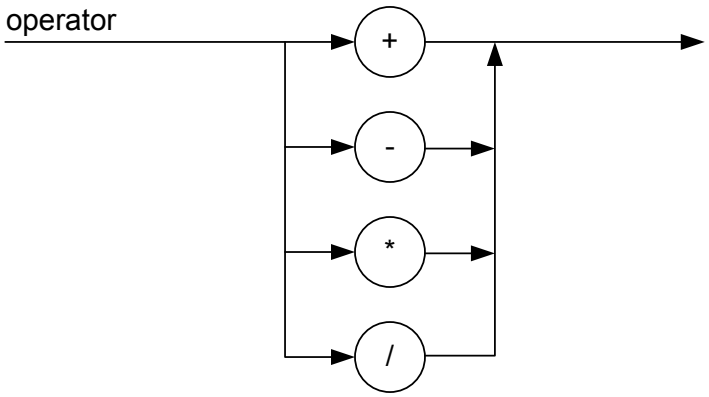
These are the basic requirements for an expression, but it is necessary to be more precise and to detail the terms that may occur. I will do so using syntax diagrams.

Overall, an expression is as follows:



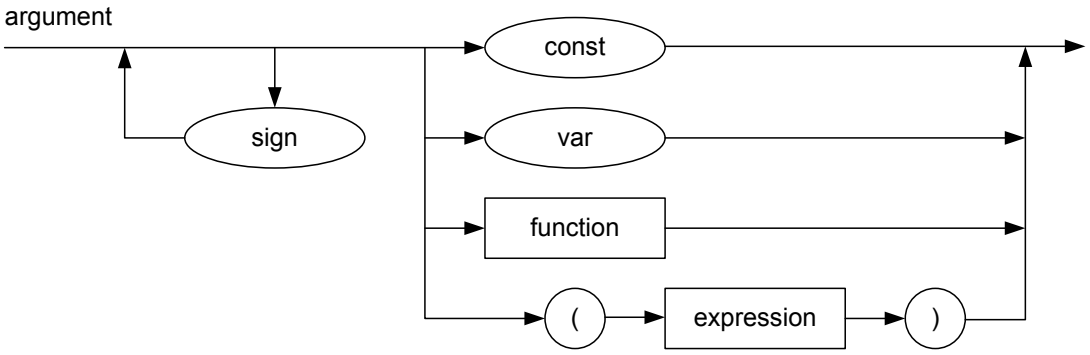
That is, an expression may be a single argument, or it may be an argument, followed by an operator, which in turn is followed by a new expression. Note that it is a recursive definition in which a concept – here an expression – that is defined by means of its self.

There remains to define what is understood by, respectively an argument and an operator. I will start with the latter, which is merely a symbol for one of the four arithmetical operations. It can also be described as follows:



Download free eBooks at bookboon.com

Then there is an argument which is a somewhat more complex concept:

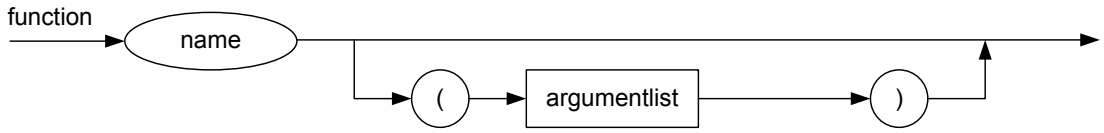


An argument can start with one or more signs (plus or minus). Then there are four possibilities

- *const*, there will always be a non negative number – a *string* that can be converted to a positive *double*
- *var*, that is a variable and thereby a string of the form *Xn*, where *n* is an index
- *function*, that is a mathematical function
- an expression in parentheses

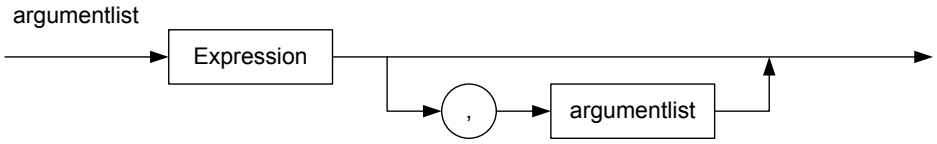
Click on the ad to read more
Download free eBooks at bookboon.com

Finally there is to define what a function can be and it can be described as follows:



and where the *name* is one of the functions mentioned above.

I also must define what an *argumentlist* is:



Note that it is again a recursive definition, and an argument list is one or more expressions separated by commas.

Thus, I have quite accurately described what we mean by a legitimate expression.

Regarding the evaluation of an expression is a matter of inserting values for the expression's variables, after then the expression's value can be determined. The value of an expression should always be a *double*.

The latter problem is the user interaction. It can be described as follows:

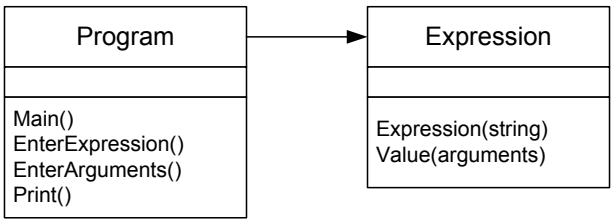
```
enter expression
as long as expression is not empty repeat
{
   if expression is syntactically correct then
   {
      enter arguments
      as long as there are entered arguments repeat
      {
         if the arguments are legal then
         {
            evaluate expression
            print the result
         }
         or print an error message
         enter arguments
      }
   }
   or print an error message
   enter expression
}
```

This means that entering either an expression or arguments ends when you just typed enter. Arguments are entered as numbers on the command line separated by spaces.

Download free eBooks at bookboon.com

Design

Basically, I think of an expression as a class, and apart from a constructor that creates the expression from a string the class should really only have a single method that can evaluate the expression. Equivalent and based on the above outline concerning the user interaction you can outline the program as:



Here is the program and thus the class *Program* in principle simple and basically consists of

- the user interaction corresponding to the outline from the analysis
- entry and parsing an expression with checking whether an expression is legal
- entry and parsing of arguments to an expression
- print the result of evaluating an expression
- error handling

Regarding the first, is the sketch from the analysis actually an algorithm for the *Main()* method, and there is not much to consider.

Entering an expression is simply entering a string, but parsing and checking whether an expression is legal is by no means simple and the whole challenge of the task. However, I will defer this task to the constructor in the *Expression* class, so it gets a string as a parameter and it is so constructor's task based on the syntax diagrams from the analysis to check whether the string represents a legal expression. If not, the constructor should throw an exception.

When an expression should be evaluated, the user must enter one or more arguments separated by spaces. Each of these arguments must be parsed into a *double*, and it must be done in the *Program* class, so the method *Value()* in the *Expression* class has as argument an array of the type *double* as a parameter.

Print() is a simple method which will print the expression, arguments and what the expression is evaluated to, for example something like

$$2*\sin(x0)+sqrt(5*x1+3)$$
$$1.41$$
$$3.14$$
$$= 6.2985498411563$$

Download free eBooks at bookboon.com

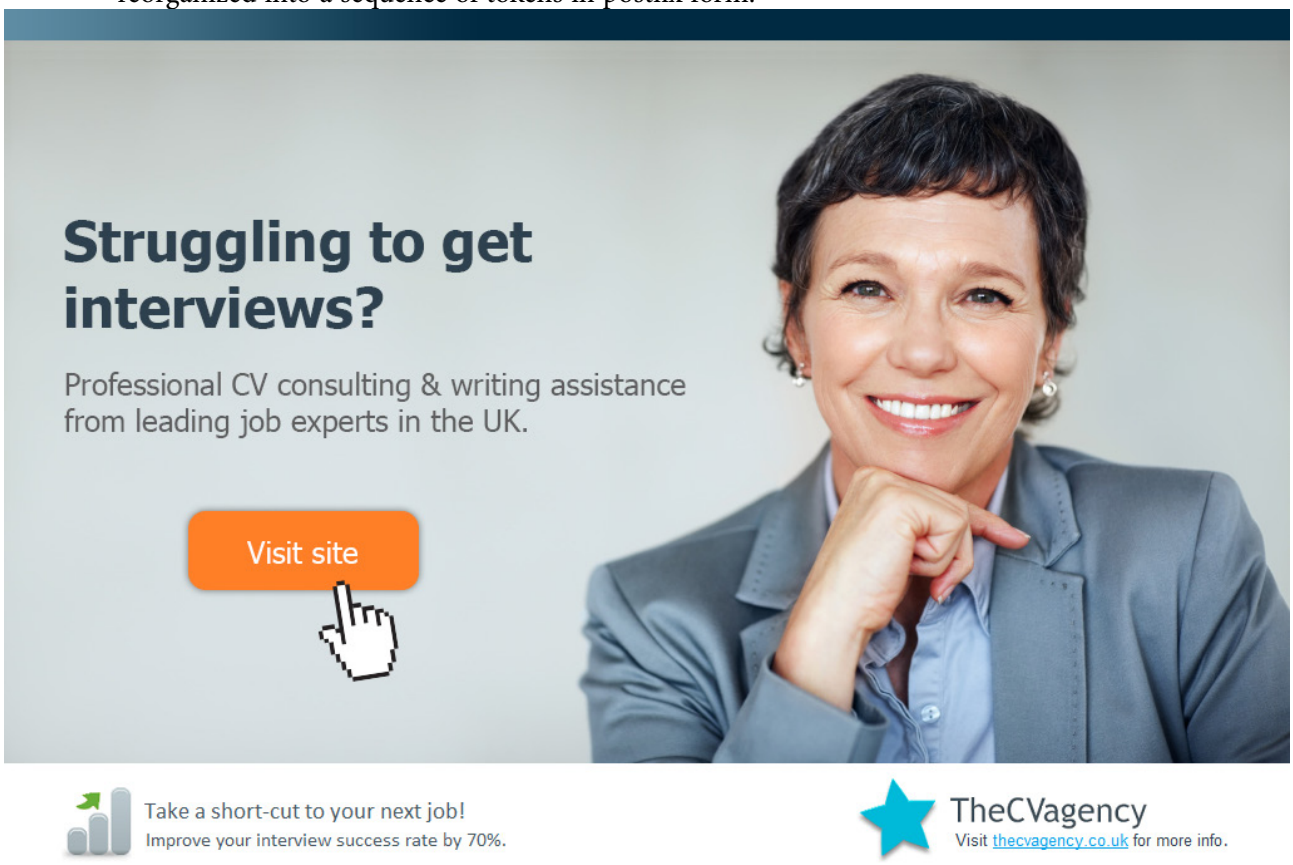
When the result is written in several lines, it is because and expresses incl. arguments can be long.

Finally, there is the error handling. When there is an error, there the program must print an explanatory error message on the screen, and in principle errors can arise in three situations:

- The user enters an expression which is not syntactically correct. In this case the constructor of the class Expression raises an exception.
- The user enters arguments to an expression that is not legal. In this case it is the class *Program* which must capture and process the error.
- An expression can not be validated because there is an error in the calculations (eg. division by 0). Here the *Expression* class must throw an exception.

In class *Expression*, there are two challenges in form of the constructor and the method *Value()*, and here the first challenge is the largest. The constructor has a parameter in the form of a string and must perform three operations:

- Scanning the expression which means that the string should be split into the elements (tokens) that the expression consists of.
- Parse the expression that means to control that the expression's syntax is correct according to the syntax diagrams from the analysis.
- Converting the expression to postfix form, meaning that the expression elements must be reorganized into a sequence of tokens in postfix form.



Download free eBooks at bookboon.com

To solve the first problem, the parameter string is divided into elements which are usually called tokens. A token is thus an object that indicates an element that can be part of an expression. If you for example consider the expression

$$(123+56)*789$$

it consists of 7 tokens:

(123 + 6) * 789

When the string is divided you have a number of tokens, each of which is a string. The individual tokens must be treated differently and have different properties, so I will write a class to each token that can occur, and the classes are arranged in a hierarchy (see next page). All types are exceedingly simple, and the cause of the hierarchy is partly that it allows one to treat all tokens in the same way and second, that in that way it is easy to expand with new tokens, if such an expression must support a new function.

After the scanning, you have a list of tokens, and parsing has to investigate whether this list of tokens is in accordance with the syntax rules. In the code it corresponds to write a method corresponding to each of the above diagrams, and it is in principle simple, but requires a technique called *recursion*, that is described below.

Usually you write an expression on infix form which means that one writes the operator between two operands, ie for example.

$$2+3$$

which means the sum of 2 and 3. If there are several operators, we need rules for how the expression must be evaluated. For example means

$$2*3+4$$

that you first calculate the product of 2 and 3 and then adds this result to 4 – the result is 10. In contrast, means

$$2+3*4$$

that you first calculates the product of 3 and 4, since one attaches multiplication higher precedence than addition – the result is therefore 14. If you wish to suppress this rule, so the addition is first performed, it is necessary to insert parentheses:

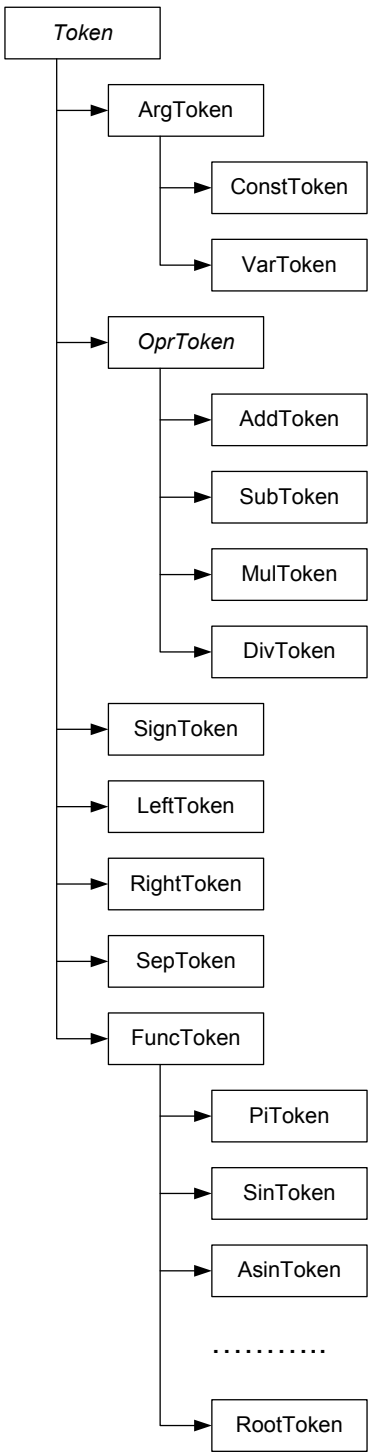
$$(2+3)*4$$

having the value of 20.

Download free eBooks at bookboon.com

If in an expression there are several operators of equal priority, the rule is that the operators are evaluated from left. Below is first computed the sum of 2 and 3 and then subtract 4 because addition and subtraction have the same priority:

$$2+3-4$$



Download free eBooks at bookboon.com

An expression may be more complex, for example

$$(1+2*(3+4)\ /((5+6)*7)$$

where there are parentheses within the parentheses. The value of the expression is, moreover, 0.194805. When an expression contains parentheses, the parentheses are evaluated first, starting with the innermost parentheses. It is certainly possible to write a method that does it, but if the expression becomes more complex with mathematical functions and many parentheses, it is not simple, and therefore one will typically go on a second path and instead convert the expression to postfix form. This means that an operator is assigned in accordance with the arguments that it should operate on. Thus, the above expression is written as

$$23+$$
$$23*4+$$
$$234*+$$
$$23+4*$$
$$23+4-$$
$$1234+*+56+7*/$$

Click on the ad to read more

Download free eBooks at bookboon.com

If for example you must calculate the value $23*4+$, it works as

$23*4+$
$64+$
10

The idea is that any expression can be written in postfix form without using of parentheses. When the expression should be evaluated, it is traversed just from left to right. Every time you come to an operand, put it on a stack. Coming to an operator you pop the stack twice (if it is an operator with two arguments) calculates the result and put it on the stack. Finally the stack will contain only one element which is the result. The method may based on the last of the above expressions be is illustrated in the following manner:

			4									
		3	3	7				6		7		
	2	2	2	2	14		5	5	11	11	77	
1	1	1	1	1	1	15	15	15	15	15	15	0.1948
1	2	3	4	+	*	+	5	6	+	7	*	/

The conclusion is that it is much easier to evaluate an expression in postfix form than one in infix form and it is therefore worthwhile to seek a strategy (an algorithm) to convert an expression from infix to postfix form. It may be done in the following manner by using a stack:

```
the expresseion is traversed from left and for every element
     1.   if it is a sign push it on the stack
     2.   if it is a funtion push it on the stack
     3.   if it is a variable push it on the stack
     4.   if it is a number push it on the stack
     5.   if it is a left parenthes push it on the stack
     6.   if it is a right parenthes, then pop the stack and add the top of the
          stack to the resultat until you get a left parenthes
     7.   if it is an operator then pop the stack and add the top of the stack to
          the resultat as long as the priority of the top of the stack is less than
          or equal to the priority of the element, push the element on the stack
pop the stack and add the top of stack to the result until the stack is empty
```

As you can see, it is crucial in the algorithm that there are assigned the right priorities for the individual elements. It is the priorities that determine when to move from the stack to the result, which is just a list. The two important points in the algorithm are 6 and 7. If you get to an operator – for example a multiplication – you must first move everything on the stack with a better priority than multiplication to the result list. It will be numbers, variables and functions, and then the multiplication operator is put on the stack. So, if the expression tokens are converted to postfix form, it is simple to implement method *Value()* in the class *Expression*.

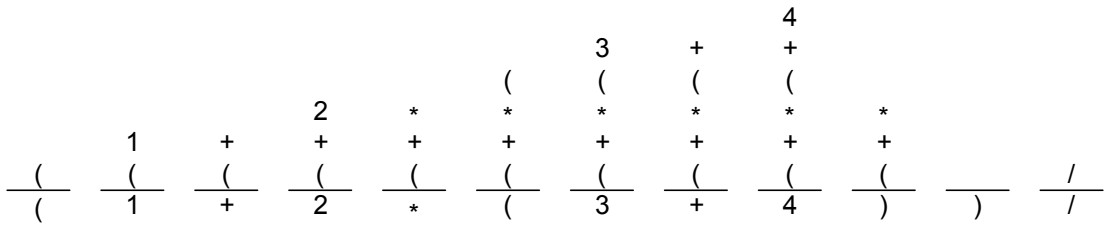
Download free eBooks at bookboon.com

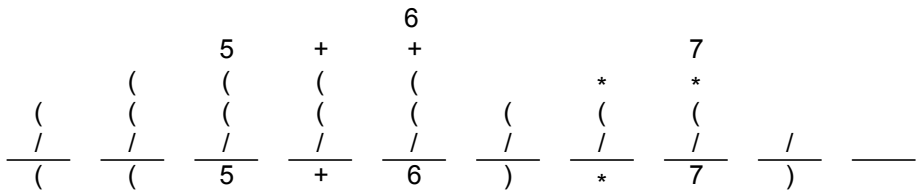
If you look at the expression

$$(1+2*(3+4)\ /((5+6)*7)$$

it can be converted to postfix form in the following manner:

```
1
1 2
1 2 3
1 2 3 4 +
1 2 3 4 + * +
1 2 3 4 + * + 5
1 2 3 4 + * + 5 6 +
1 2 3 4 + * + 5 6 + 7 *
1 2 3 4 + * + 5 6 + 7 * /
```





In this particular task, I will apply the following priorities:

- numbers, variables 0
- sign 1
- function 2
- multiplication, division 3
- addition, subtraction 4
- left parenthes 9
- right parenthes, comma 99

Download free eBooks at bookboon.com

Comment

To write the code (syntax checking) I need recursion and therefore a few words about what it is.

A method in a class can be seen as an isolated code that performs a specific operation on the basis of parameters and possibly returns a value. The method's statements are the commands it performs, and there are no limitations on what it can be. It may, for example be calling another method, and a method thus especially can also call itselves. If so, one says that the method is recursive. As an example of a recursive method – and a method which does not relate to the specific task – is shown a method that determines the factorial of *n*:

```
static ulong Factorial(uint n)
{
   if (n == 0 || n == 1) return 1;
   return n * Fakultet(n - 1);
}
```

At first glance, recursive methods are difficult to comprehend, but if first one is familiar with the principle, it is not particularly difficult. Above if *n* is 0 or 1, one can directly determine the result. If *n* is greater than 1, one can determine the factorial of *n* as *n* times the factorial of *n-1*. One can think of it in that manner that to determine the factorial of *n* is reduced to determining the factorial of *n-1* which is a smaller problem than the starting problem: To determine the factorial of *n*. If you repeat that operation a sufficient number of times, to get to the simple case where *n* is 0 or 1 and where one can directly determine the result.

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

Click on the ad to read more

Download free eBooks at bookboon.com

The principle of a recursive method is such that a problem may be divided into two problems: A simple problem which can immediately be overcome, and a simpler (smaller) problem, of the same kind as the original.

Formally, the factorial of *n* is defined as follows:

$$n! = \begin{cases} 1 & for\ n = 0\ or\ n = 1 \\ n * (n-1)! & for\ n > 1 \end{cases}$$

It is by a recursive definition, and in such situations is recursion often a good solution. In this case, the method *Factorial()* is simply only a rewrite of the mathematical definition to a method in C#.

It is clear that in this case, the method also can be written iteratively by means of a simple loop, and this solution would even be preferable, while in other situations, the recursion is a good solution that can provide simple solutions and even a code that is easier to read and understand than a corresponding iterative solution.

However, there are reasons to always be aware of recursive methods because each recursive call creates an activation block on the stack. There is therefore a danger that the recursive methods use the whole stack, with the result that the program will crash.

Programming

This time there are no changes for either the analysis or the design, and the program is written entirely in accordance with the above design.

According to the task, there were two specific requests:

- That it should be easy to expand an expression with new functions.
- That an expression could be used in contexts other than the specific task.

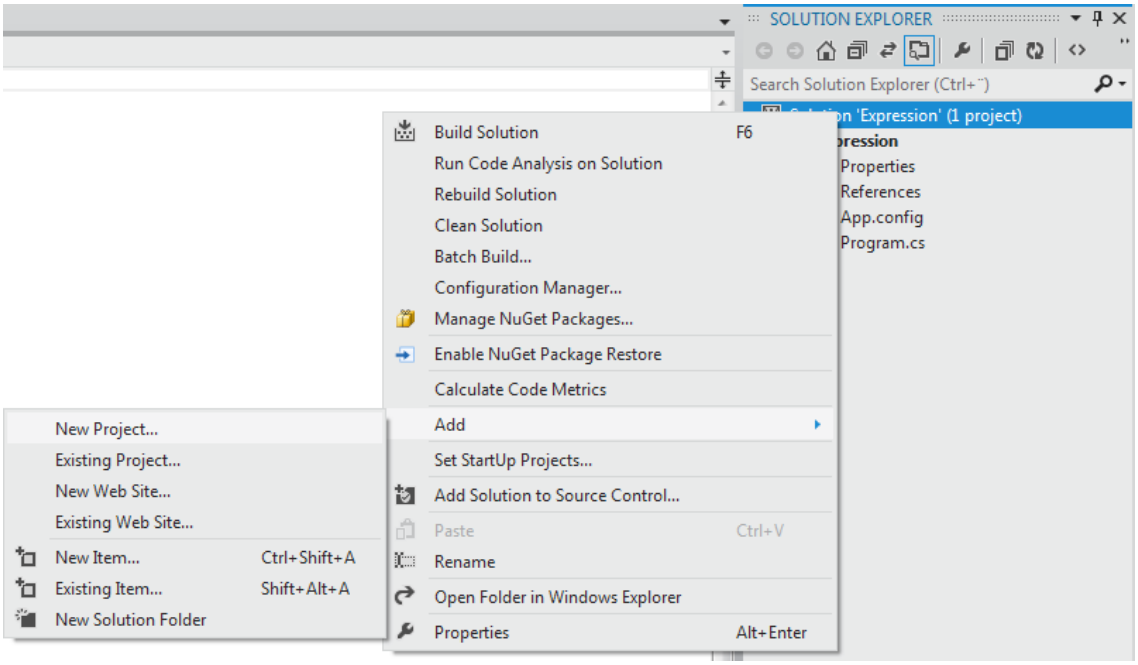
I will therefore examine to what extent these goals have been met.

The program consists primarily of a class *Expression*, combined with a large number of other types in terms of interfaces and classes for the individual tokens. Finally, there is a static class *Tokens* with only one method, which will translate the name of a token to a *Token* object, as well as a static class *Tokenizer* with two static auxiliary methods to scan the input string.

In order to better reuse these classes in other contexts, they are placed in a class libray, and the result is a dll that is used in the main program – and thus in the same way can be used in other applications that may need the type *Expression*. Since I only once previously have shown how to make a class libray I will go through the process.

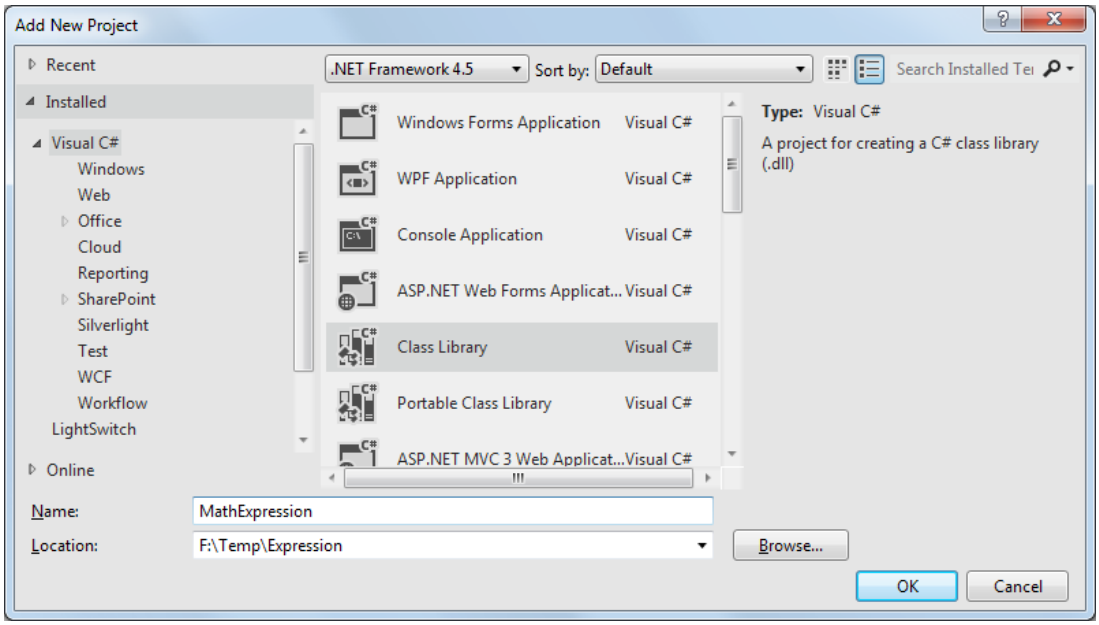
Download free eBooks at bookboon.com

I'll start with a new *Console Application* project that I've called *Expression*. This creates a *Solution* named *Expression* and including is a project also called *Expression*. A *Solution* in Visual Studio can have multiple projects, and in this case I right click on the *Solution* name in *Solution Explorer* and select *Add* | *New Project*:



Click on the ad to read more
Download free eBooks at bookboon.com

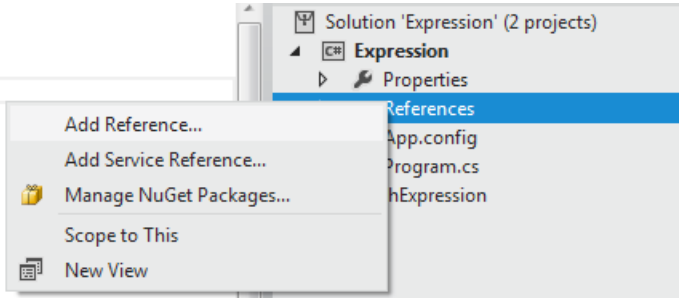
Then you get the usual project window, and I create a *Class Library* project, which I have called *MathExpression*:



I delete the auto-generated class called *Class1*, and then adding and writing the classes as my library should contain.

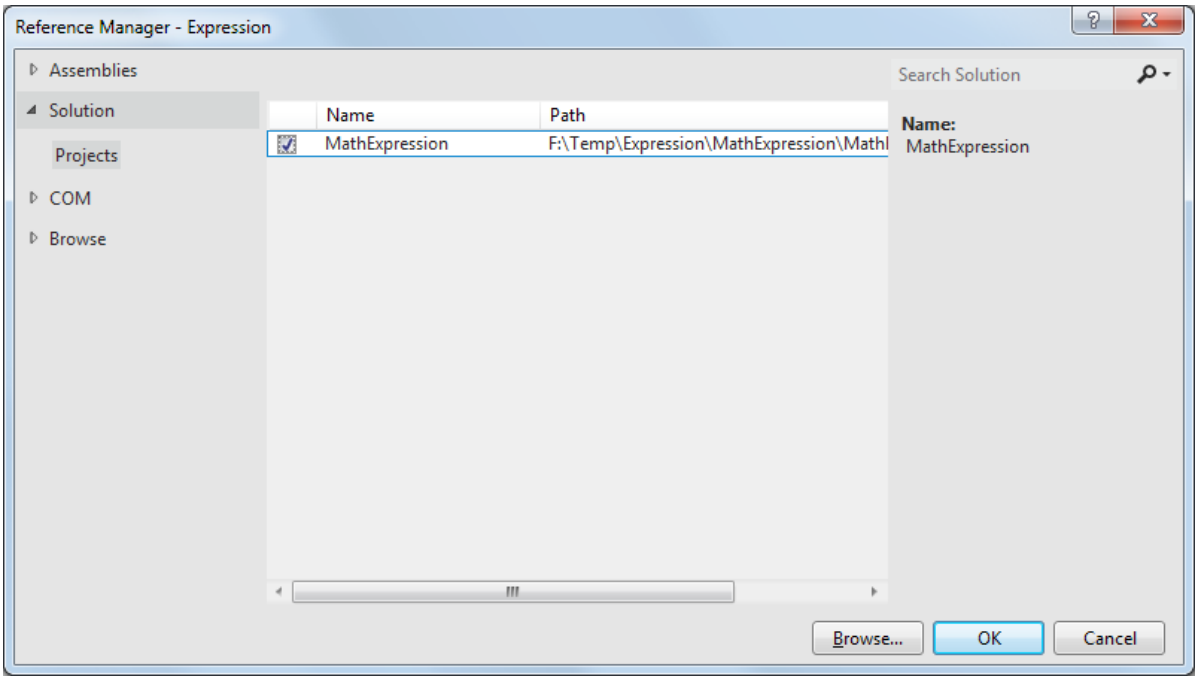
I made another little step in terms of creating a reusable class library. The class *Expression* is the primary class, and it is to be used by the program itself and must therefore be defined public. The same applies to the class *Tokenizer* since it also may have an interest in contexts other than the current program, but all the other classes are defined *internal* (which, incidentally, is the default), indicating that they can only be used within the dll that contains the classes. Of course it is not necessary, but it is classes, all of which can be thought of as auxiliary classes to *Expression*, and therefore should in principle not be available outside the dll. With Visual Studio terms one can think of an internal type, as a type that can only be used for the project that it is a part of.

With the dll finished you must set a reference to it in the main program. It is easy, since both projects are in the same *Solution*. In the project *Expression* right click on *References*:



Download free eBooks at bookboon.com

and here you choose *Add Reference*. You can now find the dll in the *Solution* page:



After this the dll is added to the program's project and can be used.

Then there is the problem that it should be possible to expand the program with new functions, and it is only partially solved. As an example, I will expand the class *Expression* with a function that determines a factorial. This requires two changes in the dll. First, adding a new token:

```
internal class FactorialToken : FuncToken
{
   public override int Count
   {
      get { return 1; }
   }

   public override double Value(params double[] x)
   {
      if (x.Length != Count || x[0] < 0)
         throw new ExpressionException("Factorial(x), x >= 0: Ulovligt argument...");
      uint n = (uint)x[0];
      ulong u = 1;
      for (uint i = 2; i <= n; ++i) u *= i;
      return u;
   }

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


Download free eBooks at bookboon.com

Next, add a new entry in the *switch* statement in class *Tokens*:

```
case "factorial": return new FactorialToken();
```

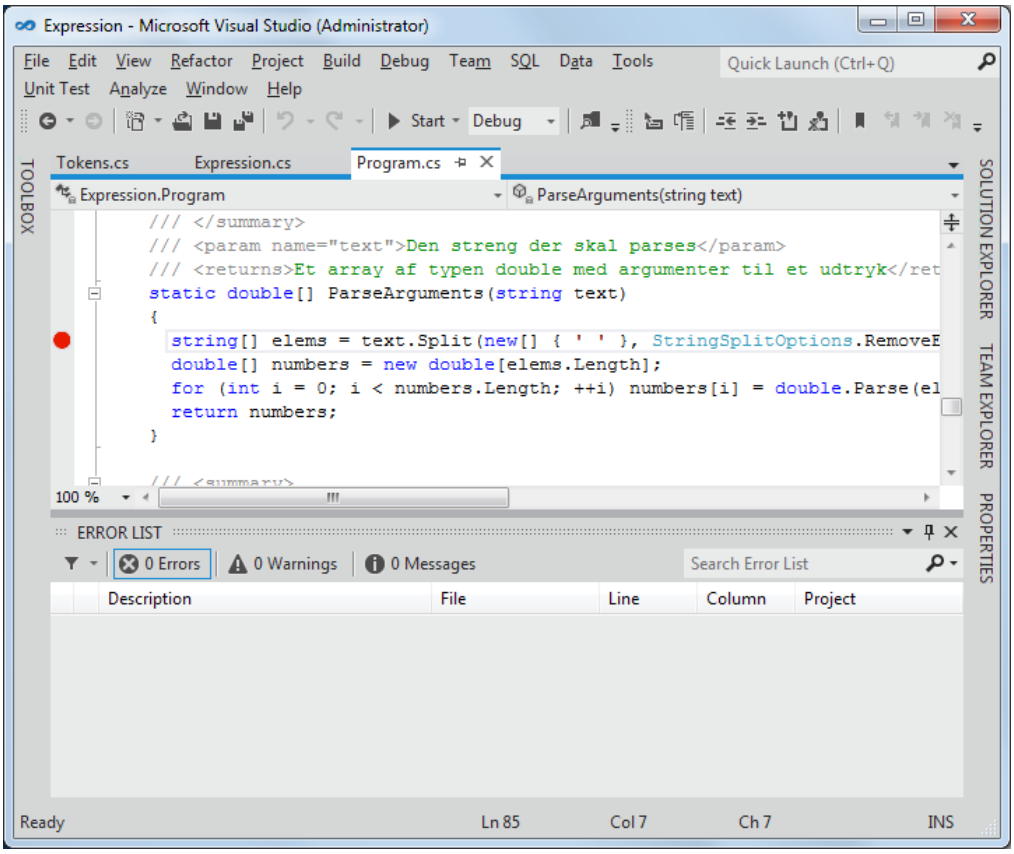
It is also far and *Expression* now has a new function. Note that it is necessary to modify the code, but only in two places and only in the internal classes, and the code of the class *Expression* is unchanged.

Comment

To conclude this book, I will mention the *Debugger*, which is an important tool in Visual Studio. The following is by no means a complete review of the debugger, but it is a hint of what it can do.

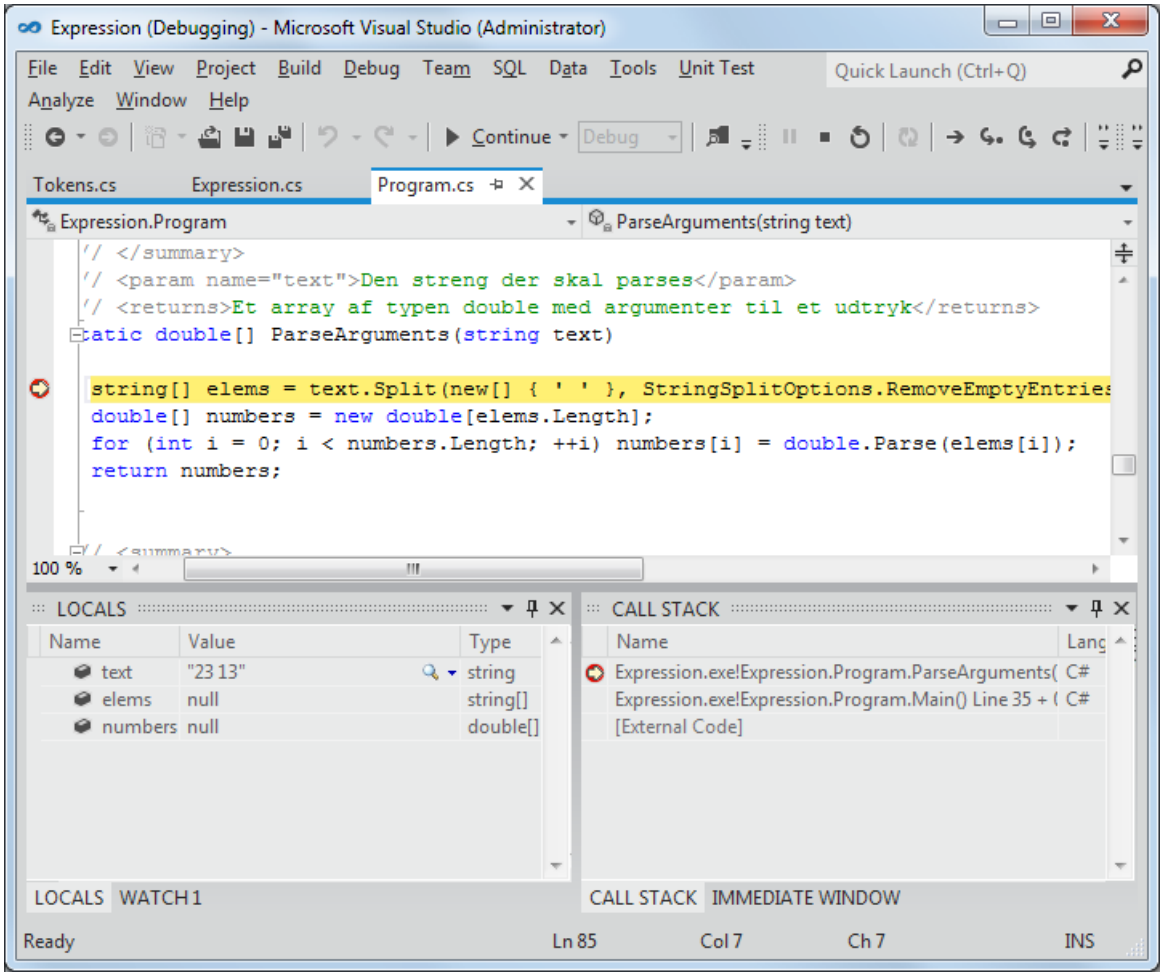
Once you've written a program and will test it, it will often be such that it fails. Maybe you get a wrong result, or the program crashes. The task is then to find the error and correct it, and here the debugger can help. You can set a breakpoint, which means that the running stops when the program reaches this point in the code, and you can then see the value of variables and examine whether they have the right value. One can also go forward in the code statement for statement and all the time follow what happens with the variables. The debugger is an excellent tool to find where a program fails, but generally the debugger can be used to analyze the code.

As an example this application has a method *ParseArguments()*, which is used to parse the arguments for the expression. Here I put a breakpoint in the method's first statement, which you do by simply clicking the mouse:



Download free eBooks at bookboon.com

If you now run the program as *Start Debugging*, enter an expression and then enter the arguments for the expression the run will stop on the line where there is set a breakpoint. At the same time one can see the local variables and parameters and their values. One can then by pressing F10 go through the program and see what happens with variables. By selecting the tab *Watch* is also possible to keep track of instance variables.



Download free eBooks at bookboon.com