Poul Klausen

C# 4

Important Classes and More

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 4: IMPORTANT CLASSES AND MORE

SOFTWARE DEVELOPMENT

C# 4: Important Classes and More: Software Development
1st edition

ISBN 978-87-403-3465-4

CONTENTS

FOREWORD

The previous book deals with object-oriented programming in C# and including the core concepts such as classes and interfaces, and this book is in many ways a continuation such that the book describes a number of topics that were not accommodated in the previous book. The book describes generic types and in relation to that collection classes, among other things, but there are also a number of concepts that are basic prerequisites for C# programmers and including delegates and lambda and other concepts that may not be absolutely necessary but useful to know. After reading this book and working on the book's examples and exercises, you should have a good background for writing programs with C# as your programming language.

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

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

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

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

1 INTRODUCTION

C# is a comprehensive programming language with many concepts. It is an object-oriented programming language and the most important concepts related to object-oriented programming are described in the previous book. However, there are a number of other programming concepts that are important and that are the subject of this book. Here I would like to mention in particular:

1. generic types and methods
2. collection classes
3. delegates
4. lambda

and maybe also operator overriding.

Generic types are types where you by a parameter indicates the kind of other objects that objects of the generic type must contains. As an example you can think on a list where you by a parameter like *List<T>* defines which type of objects the list should contain. C# has many other generic types, but you can also define your own generic types as is explained in the following.

List<T> is a generic type and a type which I have used many times, but it is also an example of a collection type and then a type that is a container for other objects. It is only one example on a collection type, and there are others. The collection types differs in terms of how they store there objects and which services as methods they make available, and both have a decisive influence on which collection class to choose in a specific case. The collection classes each have their own purpose and it is therefore important that you know the differences.

I have already used delegates many times, especially for applications with a graphical user interface where delegate are used to define events. A delegate is a reference type and as so you can define variables, parameters and others whose type is a delegate. Basically, a delegate defines a type for a method and therefore allows one to pass a method as a parameter to another method. In the previous examples, I have used delegates without explaining exactly the term, and although it may sound simple, there are actually some details. Therefore, delegates have their own chapter in this book.

C# is as mentioned an object-oriented language, but lambda is a step towards introducing functional programming in C#. It does not means that C# is a functional programming language (a language in which everything is done by calling and composing functions), but it means that you better can to use functional syntax where it seems natural. You can live without lambda, but when you come to LINQ you can't, and today lambda should be seen as an important part of C#.

In the end, there is operator overriding which also is something you can live without depending on the tasks you have (the programs you have to write), but conversely there are also tasks where operator overrides can contribute to a good solution. In short, it is all about assigning most operators a function for custom types as well.

2 STRINGS

The type *String* is a class, although in most cases you uses a *String* as the other simple data types. In most cases, the difference is not of great importance, but in some contexts it is something you should be aware of. *String* is defined in the namespace *System*, and the compiler has a reserved word for the class called *string*. This means where you writes *String, System.String* or *string* it means the same. In the previous books I have used strings many times and I think in all examples, and there is not much else to mention about the type *string* than what else already is told, but below follows a few things you should note.

The class *String* is *sealed*, which means the class cannot be inherit. Note (which has nothing with the class *String* to do) that also a method can be sealed. This means that the method cannot be overridden in a derived class. This can be used, if a method is defined *virtual* in a base class and overridden in a derived class you can then define the overridden method as *sealed* to prevent the method to be overridden in another derived class.

You should be aware that the class has a number of methods to manipulate strings, and it pays to investigate which methods are available. Many of them I have used already and more are used in the course of the books. The class also have useful static methods. I will not discuss these methods here, but they will be used in the books examples, and I've already used many of these methods.

The class implements the interface *IComparable*, and as so the class implements the method

```
CompareTo(String str)
```

which is the comparison method for strings that compares strings in alphabetical order. When the class *String* implements this interface, it means among other things that strings can be sorted with C#'s sorting methods.

An important property of the class *String* is that it is *immutable*, and that means that have you created a *String*, its state cannot be changed. For instance you cannot change a character in a string. If you, for example have the string

```
string s = "abcdefg";
```

and you want to change the character *d* to a big *D*, you must write something like the following:

```
s = s.Substring(0, 3) + "D" + s.Substring(4);
```

Here, you take a sub-string consisting of the first three characters and concatenates it with string consisting of the character *D*. This result is then concatenated with the sub-string consisting of all characters from index 4 to the end of the string. Concatenation of two strings create a new *String* object, and the above statement will create two objects, and the variable *s* is set to refer to the result object instead of the original string. At the class in this way is immutable sounds complicated, and it is at times too, but the reason is performance, where it is important that the creation of string is effective and a string not fills more than necessary. In practice it is not something you think much about when the compiler largely treats strings as other simple types, and as the *String* class has many methods.

2.1 STRINGBUILDER

When you needs many string operations the result can be a performance problem because the class *String* is immutable. Every operation results in creating a new *String* object. To solve this problems there is a class *StringBuilder*, which is a class where you can manipulate the individual characters in a string, and represents a string that can be expanded without the need to create a new object. In principle, a *StringBuilder* is the same as a *List*, but simply a list where the elements are of the type *char*. As an example the following program creates a *string* consisting a number of the letter A by adding one character at a time. It happens in two ways, where the first method *CrteateStr1()* use string concatenation while the other method appends characters to a *StringBuilder*.

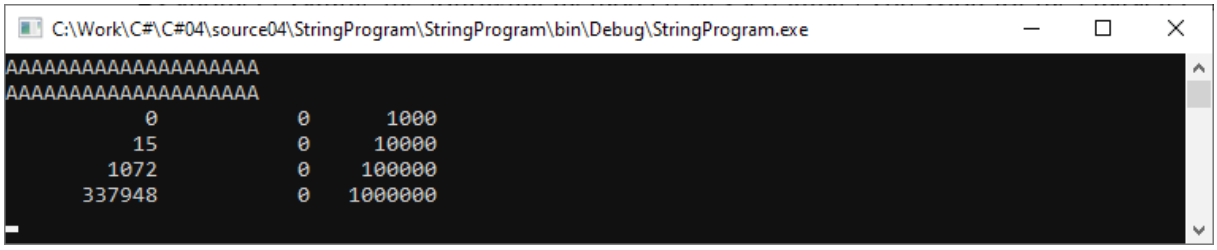
```
namespace StringProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(CreateStr1(20));
      Console.WriteLine(CreateStr2(20));
      for (int n = 1000; n <= 1000000; n *= 10)
      {
        long t1 = GetTime();
        CreateStr1(n);
        long t2 = GetTime();
        long t3 = GetTime();
        CreateStr2(n);
        long t4 = GetTime();
        Console.WriteLine("{0, 12}{1, 12}{2, 10}", t2 - t1, t4 - t3, n);
      }
    }

    private static string CreateStr1(int n)
    {
      string str = "";
      for (int i = 0; i < n; ++i) str += "A";
      return str;
    }

    private static string CreateStr2(int n)
    {
      StringBuilder str = new StringBuilder();
      for (int i = 0; i < n; ++i) str.Append("A");
      return str.ToString();
    }

    public static long GetTime()
    {
      DateTime dt = DateTime.Now;
      return dt.Hour * 3600000L + dt.Minute * 60000L + dt.Second * 1000 +
        dt.Millisecond;
    }
  }
}
```

The program has a method *GetTime()* that reads the hardware clock and returns the number of milliseconds from start of the day to the current time. The method is quite simple as the class *DateTime* which represents a date and a time has the necessary services.

The two methods *CreateStr1()* and *CreateStr2()* are also simple, and you should primary note the use of the class *StringBuilder* in the method *CreateStr2()*. The class class has a method *Append()* used to add text to the end of the *StringBuilder*, and this method is overridden for the standard types, and in this case it is used for the type *string*, but I could also have written:

```
str.Append('A')
```

The method *Main()* has a loop and for each iteration it reads the current time, executes *CreateStr1()*, reads the current time again, and the it performs the same, but this time for *CreateStr2()*. After each iteration the program prints how many milliseconds it has taken to create the two strings. If you run the program the result could be:



For small strings there is not the big difference, but if you create strings with 100000 characters it has taken almost 1 second to create the string using string concatenation, where using a *StringBuilder* the time is not measurable (in milliseconds). For a string with 1000000 characters the time is for string concatenating about 6 minutes while a *StringBuilder* still is not measurable.

This example is extreme, but you should note that if you need a lot of string operations, the difference is not only significant but can be crucial to the performance of the program. With many string operations (and it may be more than 10) you should always in examples like the above consider using a *StringBuilder*.

EXERCISE 1: STRING OPERATIONS

Create a console application project that you can call *StringProgram*. Add the following class:

```
public static class Str
{
  /// <summary>
  /// Method which cuts off a string of specific length n.
  /// If the string length is less than or equal n, the operation is
  /// ignored and the string is returned unchanged.
  /// </summary>
  /// <param name="s">The string that must be cut off</param>
  /// <param name="n">The length of the resulting string</param>
  /// <returns>The string cut off to length n</returns>
  public static string Cut(this string s, int n)
  {
    throw new NotImplementedException();
  }

  /// <summary>
  /// Method as left adjusts a string in a field of width n.
  /// If the string length is greater than n,
      the operation is ignored and the
  /// string is returned unchanged. Else the
      string is padded with the char c.
  /// </summary>
  /// <param name="s">The string needs that has to be adjusted</param>
  /// <param name="n">The width of the field</param>
  /// <param name="c">The padding char that field
      has to be filled with</param>
  /// <returns>The string left adjusted in a field of width n</returns>
  public static string Left(this string s, int n, char c)
  {
    throw new NotImplementedException();
  }

  /// <summary>
  /// Method as right adjusts a string in a field of width n.
  /// If the string length is greater than n,
      the operation is ignored and the
  /// string is returned unchanged. Else the
      string is padded with the char c.
  /// </summary>
  /// <param name="s">The string needs that has to be adjusted</param>
  /// <param name="n">The width of the field</param>
```

```
/// <param name="c">The padding char that field
    has to be filled with</param>
/// <returns>The string right adjusted in a field of width n</returns>
public static string Right(this string s, int n, char c)
{
  throw new NotImplementedException();
}

/// <summary>
/// Method as center adjusts a string in a field of width n.
/// If the string length is greater than n,
    the operation is ignored and the
/// string is returned unchanged. Else the
    string is padded from left annd
/// right with the char c.
/// </summary>
/// <param name="s">The string needs that has to be adjusted</param>
/// <param name="n">The width of the field</param>
/// <param name="c">The padding char that field
    has to be filled with</param>
/// <returns>The string center adjusted in
    a field of width n</returns>
public static string Center(this string s, int n, char c)
{
  throw new NotImplementedException();
}

/// <summary>
/// Method which remove all characters with a certain value from a
    string.
/// </summary>
/// <param name="text">The text where to remove characters</param>
/// <param name="ch">The character to be removed</param>
/// <returns>The string text after all characters with the value ch
/// are removed</returns>
public static string Clear(this string text, char ch)
{
  throw new NotImplementedException();
}

/// <summary>
```

```
  /// Method which splits a string into tokens
      where the array sep contains strings
  /// used to separate tokens. The method returns
      a list with the tokens found and
  /// two bool parameters indecates where empty
      tokens must be intepreted as tokens
  /// and where the separator strings should be intepreted as tokens.
  /// </summary>
  /// <param name="text">The string to be split into tokens</param>
  /// <param name="noempty">If true empty tokens are ignored</param>
  /// <param name="astokens">If true the separator strings is used as
  /// tokens</param>
  /// <param name="sep">A list containing the tokens</param>
  /// <returns></returns>
  public static List<string> Split(this string
  text, bool noempty, bool astokens,
    params string[] sep)
  {
    throw new NotImplementedException();
  }
}
```

You must implement the 6 methods. You should note that the class is defined *abstract*. This means that the class cannot be instantiated, and since it has only static methods, it makes nor no sense. Also note that all methods are defined as extension methods.

Once you have written the class, you should test all your methods from the *Main()* method.

2.2 REGULAR EXPRESSIONS

A regular expression is a concept that is closely related to strings, and you can use regular expressions to define patterns for strings, and then you can ask if another string has one or more sub-strings that matches this pattern. In C#, regular expressions are implemented primarily with two classes, which are called *Regex* and *Match*. The classes are defined in the namespace *System.Text.RegularExpressions*. The syntax for regular expressions is relatively complex (hard to remember) and the concept is introduced most easily by means of examples using the program *RegProgram*. The following description of the program does not really have much to do with C#, but is more an introduction to regular expressions that are not part of C#, but something that is supported by C# and most other programming languages.

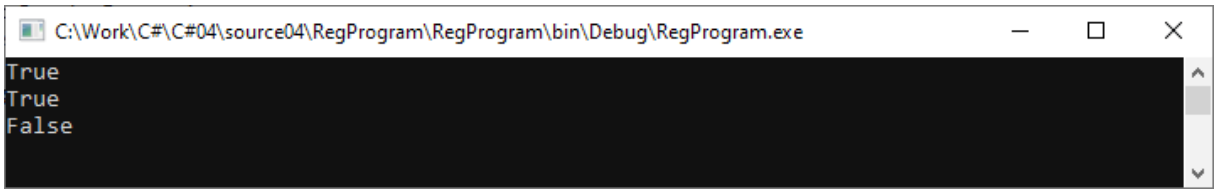
The following method has a *string* as parameter, and the method tests where the string contains the word *world*:

```
static void Test1(string text)
{
  Regex regex = new Regex("world");
  Match match = regex.Match(text);
  Console.WriteLine(match.Success);
}
```

The method creates a regular expression which is a pattern for the word *world*. It is a *Regex* object that represents a pattern which here is the word *world* and the class *Regex* defines a method used to test is a *string* contains a sub-string that matches the pattern and here the word *world* and the method returns an object of the type *Match*. If you executes the method *Test1()* as:

```
Test1("Hello world");
Test1("This world is a fine place");
Test1("Hello World");
```

the result is:



and you should note that regular expressions are case sensitive.

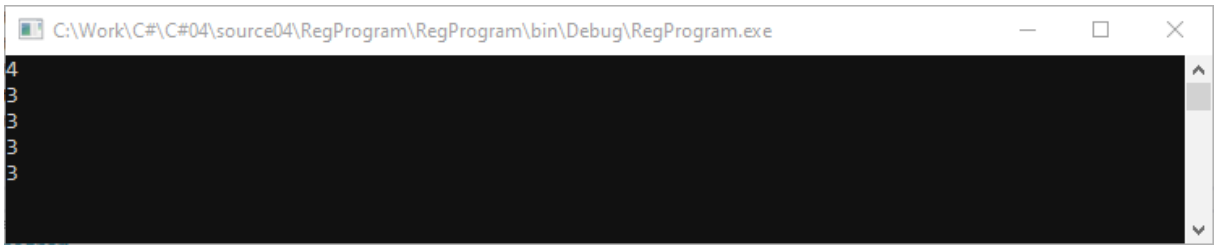
As another example the following method creates a regular expression for the character '3' that matches all sub-strings in the parameter that are "3":

```
static void Test2(string text)
{
  Regex regex = new Regex("3");
  var matches = regex.Matches(text);
  Console.WriteLine(matches.Count);
  foreach (var match in matches) Console.WriteLine(match);
}
```

The class *Regex* defines a method *Matches()* which returns a collection with objects representing all sub-strings in *text* that matched the regular expression (and here it means "3"). If you executes the method as

```
Test2("123123123123");
```

the result is:



as there are four sub-strings that matches the pattern.

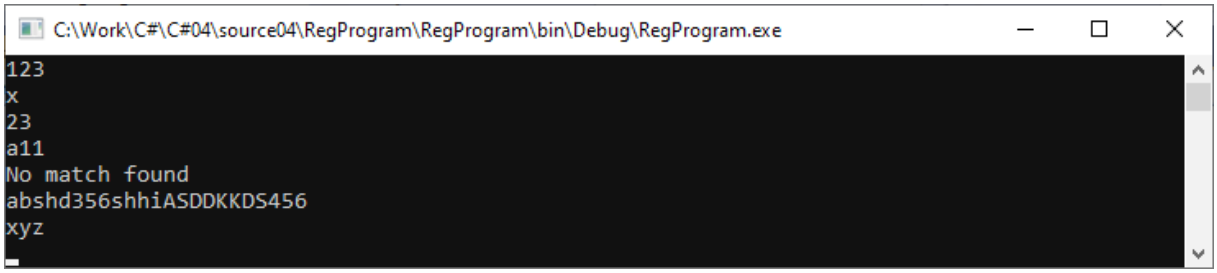
Instead of creating a *Regex* object the class has static methods to match expressions that can be used directly. In the above two examples the expression has both times simple been a *string*, but there is many other possibilities. In the next example the expression defines a pattern that matches all sub-strings consisting of the letters from A to B (both upper case and lower case letters) and all digits. To match a pattern in a string it happens in a loop over the characters and starts with an index for the first character which match and continue until the first character which does not match. The match is all characters between this two indexes. The last + means one or more matches, and there is a match if *text* contains one or more sub-strings that matches.

```
static void Test3(string text)
{
  var matches = Regex.Match(text, "[A-Za-z0-9]+");
  if (matches.Count > 0) foreach (var match in
  matches) Console.WriteLine(match);
  else Console.WriteLine("No match found");
}
```

If you executes the following statements:

```
Test3("123 + x / 23 -a11");
Test3("+-*/%<>");
Test3("-+*abshd356shhiASDDKKDS456+xyz");
```

the result is:



The first string has four matches, the next no one and the last two.

The next examples works in the same way but shows an alternate method to traverse the matches for a regular expression:

```
static void Test4(string text)
{
  Match match = Regex.Match(text, "[A-Za-z0-9]+");
  while (match.Success)
  {
    Console.WriteLine(match.Value);
    match = match.NextMatch();
  }
}
```

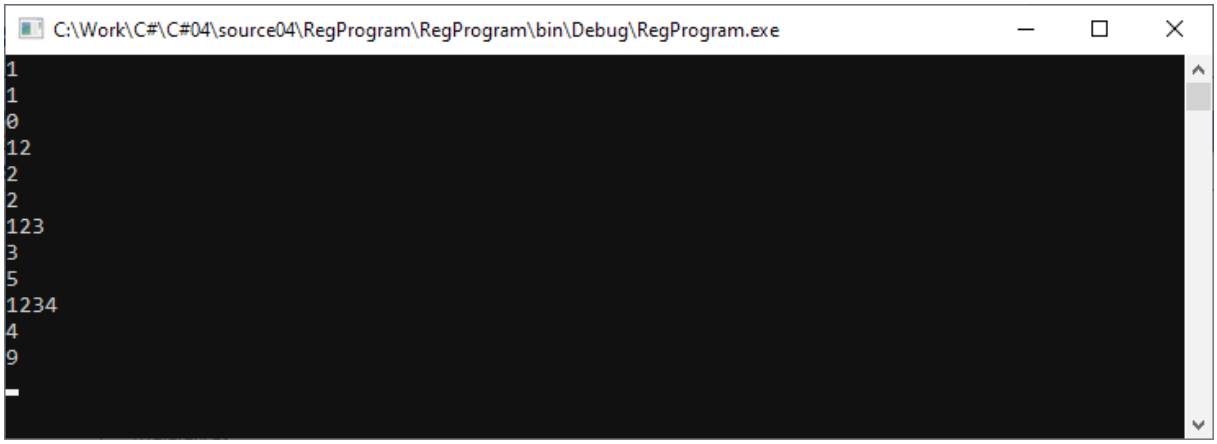
The same method is used in the following example, but it is for another expression. The expression is this time created for a pattern which matches an integer, and the pattern matches one or more integers:

```
static void Test5(string text)
{
  Regex regex = new Regex("\\d+");
  Match match = regex.Match(text);
  while (match.Success)
  {
    Console.WriteLine(match.Value);
    Console.WriteLine(match.Length);
    Console.WriteLine(match.Index);
    match = match.NextMatch();
  }
}
```

The method prints all sub-strings that matches the pattern (that is all integers in the string *text*), but for each match it also prints the length of the sub-string and there start index. If you run the method as:

```
Test5("1 12 123 1234");
```

the result is:



The last example shows that the class *Regex* has a static method to test where a string matches a regular expression:

```
static void Test6(string text)
{
  Console.WriteLine(Regex.IsMatch(text, "\\d+"));
}
```

2.3 PATTERNS FOR REGULAR EXPRESSIONS

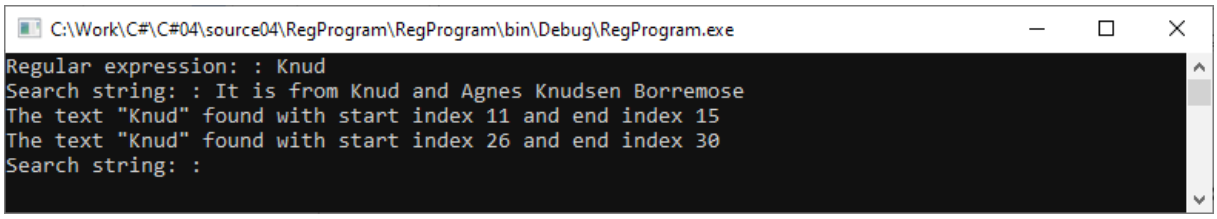
Seen from C# there nothing more to tell than what the above examples shows, but if you need to understand what options are available for patterns, there is a lot left. The program *RegProgram* has another method:

```
static void Test7()
{
  while (true)
  {
    string exp = Enter("Regular expression: ");
    if (exp.Length == 0) break;
    Regex regex = new Regex(exp);
    while (true)
    {
      string str = Enter("Search string: ");
      if (str.Length == 0) break;
      Match match = regex.Match(str);
      bool found = false;
      while (match.Success)
      {
        Console.WriteLine(
        "The text \"{0}\" found with start index {1} and end index {2}",
          match.Value, match.Index, match.Index + match.Length);
        found = true;
        match = match.NextMatch();
      }
      if (!found) Console.WriteLine("No match found");
    }
  }
}

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

It is a method that runs in a dialog with the user. It uses the method *Enter()*, which is a simple input method for entering a string. The method runs in an infinite loop. For each repetition, the user must enter a string for a regular expression, and if it is not the empty string the program creates a *Regex* object *regex* for the string that represents the regular expression. Next, the program starts an inner loop, which is also an infinite loop. Here the user must enter the string that should match the regular expression, and if it is not the empty string the program creates a *Match* object using the regular expression *regex*. This *Match* object is used to search for sub-strings that matches the regular expression. As long as there is such a sub-string, it is printed together with its start and end index.

I will use the above method to illustrate the syntax of regular expressions, and the meaning is that you should continue with your own expressions.

The simplest regular expression is simply a string that matches another string it the other string contains a similar sub-string s the matching the expression. Below is an example of a run of the above method:



That is, I as regular expression entered

```
Knud
```

while I, as a search string entered

```
It is from Knud and Agnes Knudsen Borremose
```

The result shows that the search string contains two sub-strings that matches the regular expression, and you can also see where in the search string a match is found.

To specify patterns you needs special characters, called meta-characters and has a special meaning in a regular expression. There are following characters:

```
< ( [ { \ ^ - = $ ! | ] } ) ? * + . >
```

If there is a need for these characters not to be interpreted, but is perceived as common characters in the text, you can prefix the character with a backslash.

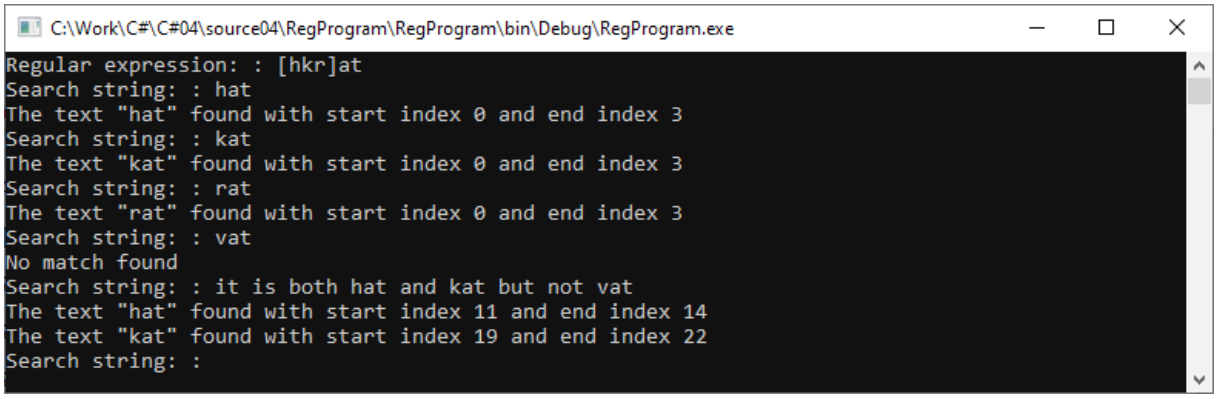
One of the basic patterns are character classes that you define as follows:

- [abc] all characters *a*, *b* and *c*
- [^abc] all characters that is not *a*, *b* and *c* (also called negation)
- [a-zA-Z] all characters from *a* to *z* and from *A* to *Z* (union)
- [a-d[m-p]] [a-dm-p]) all characters from *a* to *d* and *m* to *p* (same as
- [a-z&&[def]] all characters *d*, *e* and *f* (intersection)
- [a-z&&[^bc]] all characters from *a* to *z* but not *b* abd *c* (set difference)
- [a-z&&[^m-p]] all characters from *a* to *z* but not *m* to *p* (same as [a-lq-z])

If you, as an example, consider the regular expression

```
[hkr]at
```

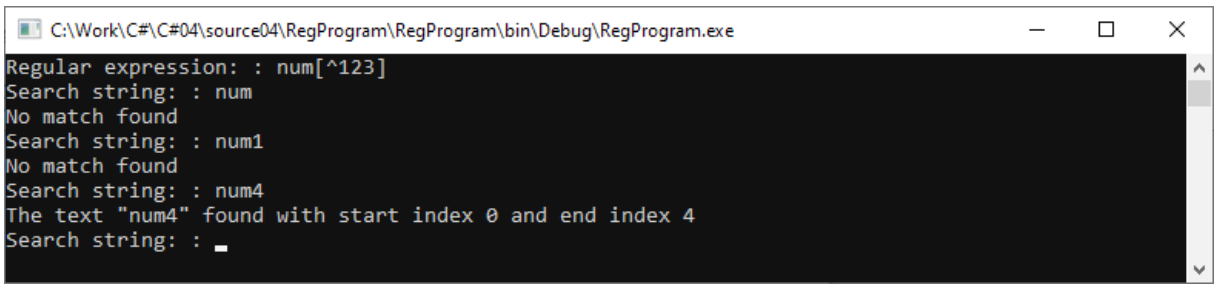
it matches the sub-strings *hat*, *kat* and *rat*:



As another example, the pattern

```
num[^123]
```

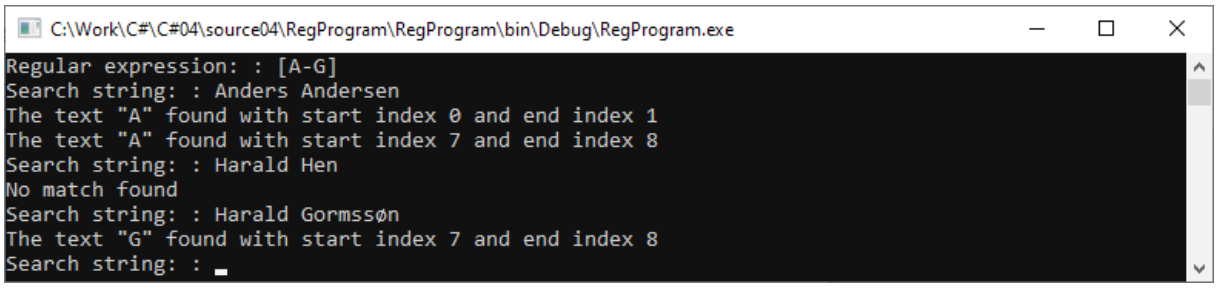
matches all sub-strings, which consists of the word *num* followed by a character which is not 1, 2 or 3:



The regular expression

```
[A-G]
```

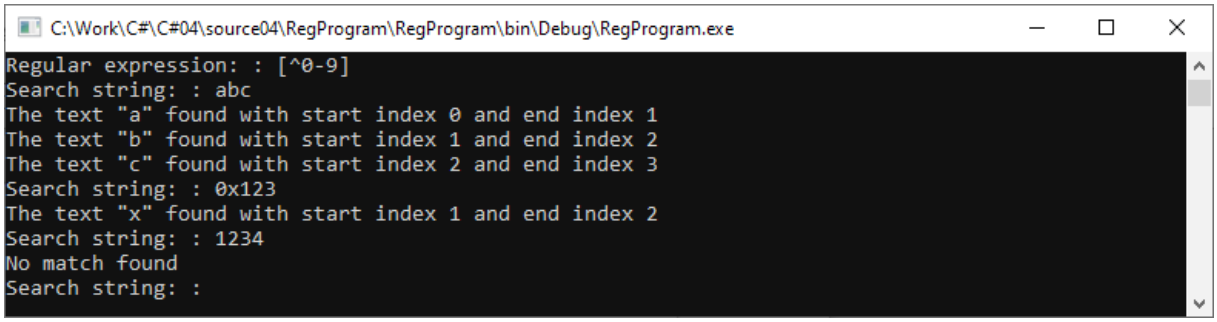
matches all uppercase letters from A to G:



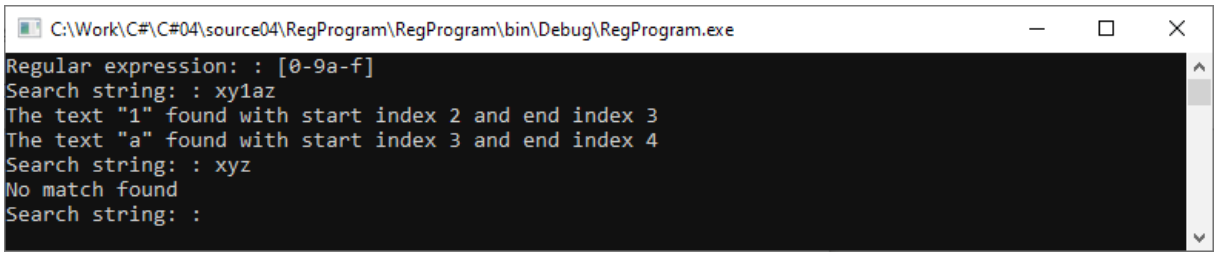
The pattern

```
[^0-9]
```

matches any character that is not a digit:



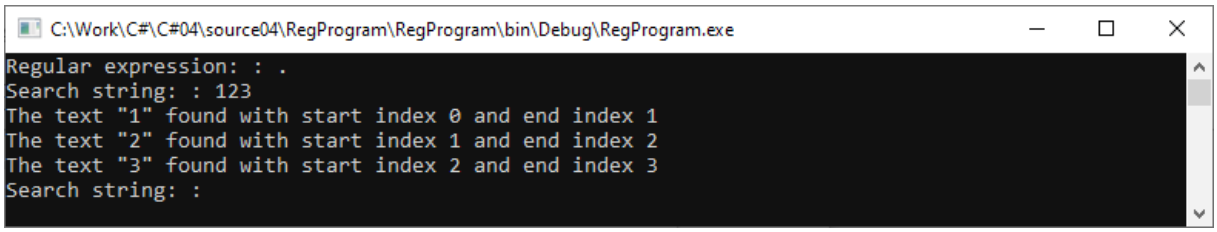
The following expression is an example of a union that matches all digits and lowercase letters between a and f:



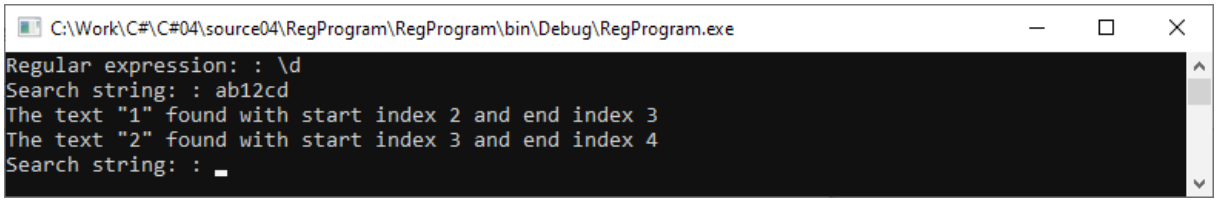
Some specific character classes can be identified by a symbol:

- . Any character
- \d The 10 digits
- \D All characters that not are a digit
- \s A *white space*: [\t\n\f\r]
- \S All characters that not are a white space
- \w Letters and digits: [a-zA-Z0-9]
- \W All other characters: [^\w]

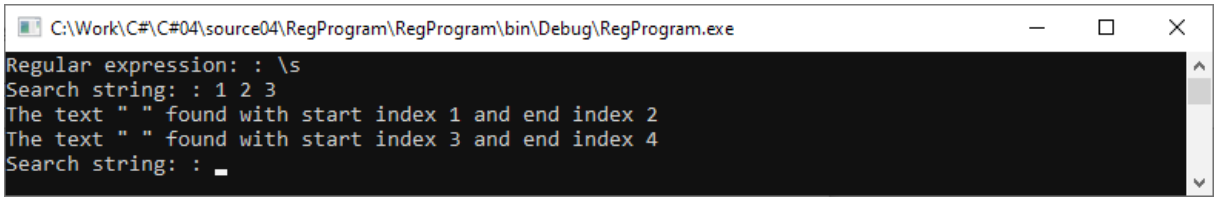
Example of a regular expression, which is a dot:



Example of a regular expression, which matches a digit:



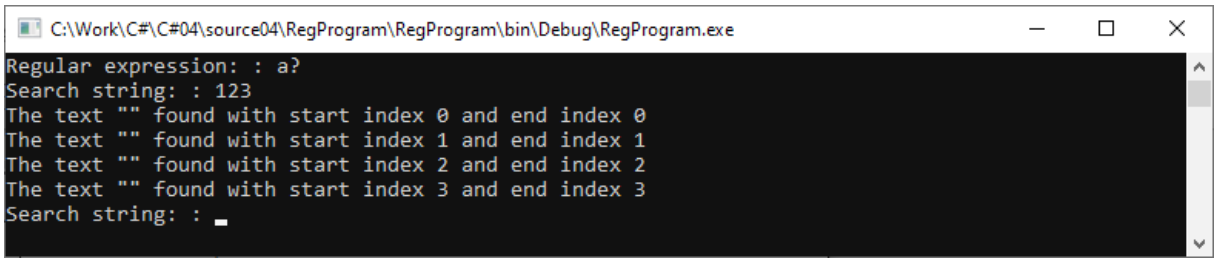
Example of a regular expression, which is a white-space:



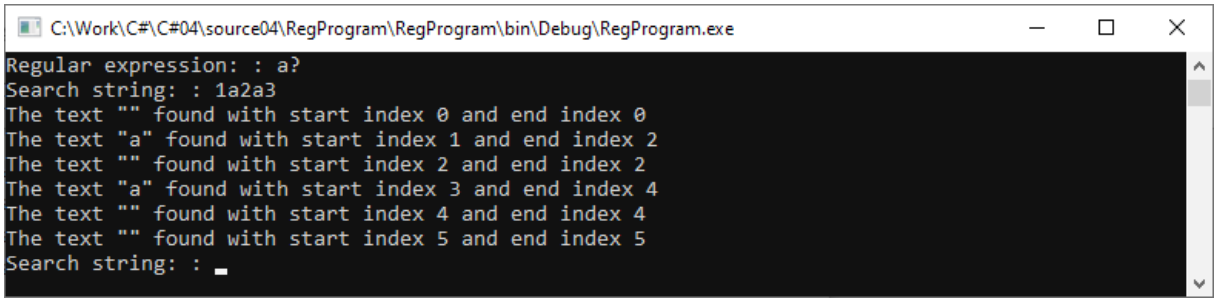
There are also some options to specify that a pattern must occur several times and for each there are even two variants. If X represents a pattern the syntax is

X?	X??	X occurs once or not at all
X*	X*?	X occurs several times or possibly not at all
X+	X+?	X occurs at least once
X{n}	X{n}?	X occurs exactly n time
X{n,}	X{n,}?	X occurs at least n time
X{n,m}	X{n,m}?	X occurs at least n times and at most m times

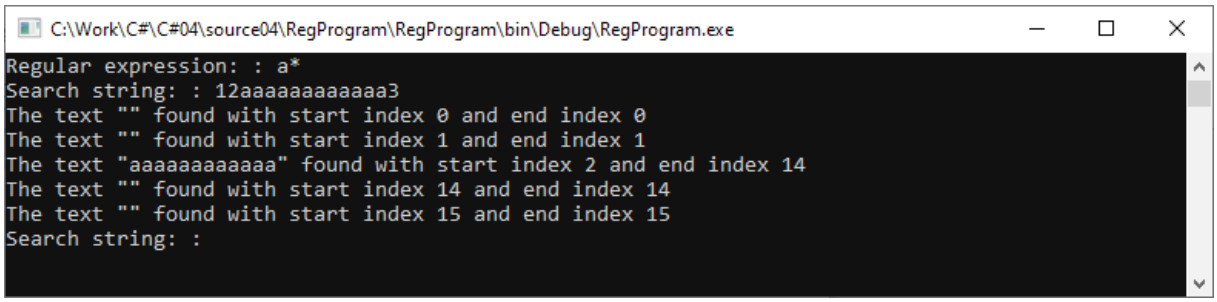
In principle, these operators are simple enough, but it is not always so easy to predict the outcome:



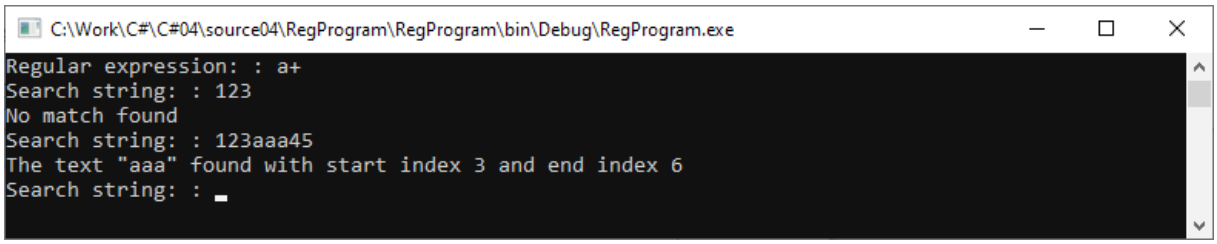
a? matches a sub-string consisting of zero or one character *a*. If *x* denotes this pattern, one can perceive the search string 123 as *x1x2x3x*, and you will therefore have 4 matches. Therefore, the following pattern results in 6 matches:



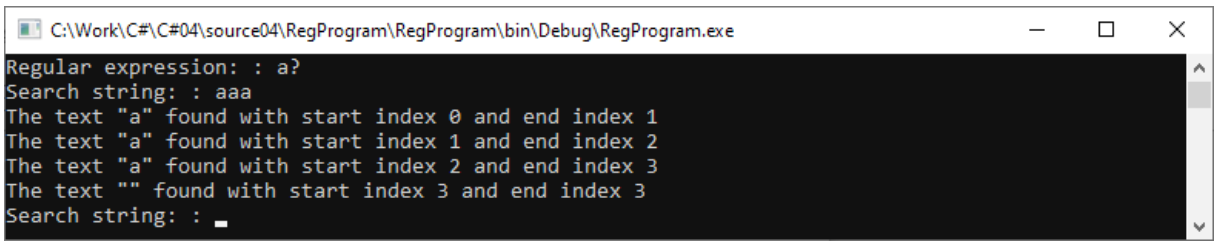
a* matches any number of the character *a*, so you end with 5 matches:



As a+ requires at least one character *a*, you below get respectively no and 1 match:

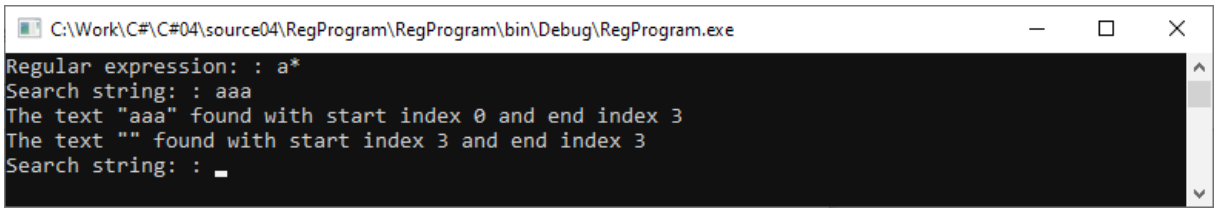


The above examples show how the method *Match* works. It searches over again in the search string until it finds a sub-string that matches the regular expression. If it finds a match, the index has reached the first character after the string. The next search will start from that location. Consider again a search similar to the above:

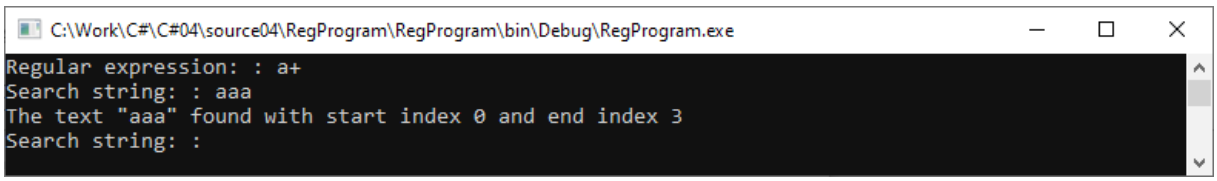


1. The search starts from the beginning, and the index is 0. The search ends when the index is 1 and the sub-string "a" matches the regular expression.
2. Next search stops with the index in place 2, where the search has found the next sub-string "a" that matches.
3. The third search finds the sub-string "a" and stop index of 3.
4. Finally stops the last search when the end of the string is reached, and at that time the index is still 3 and you have found the empty string that matches the regular expression.

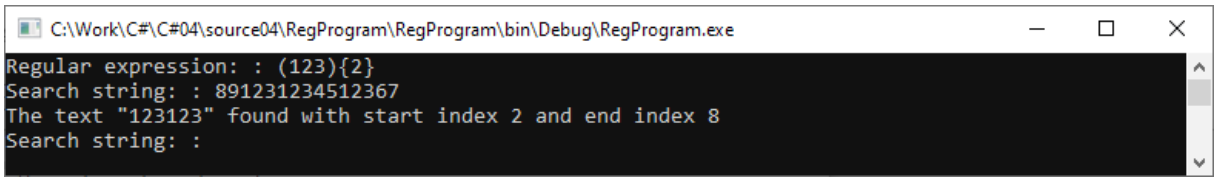
Below is an example that shows a bit of the same, but the first search stops first when the index is 3:



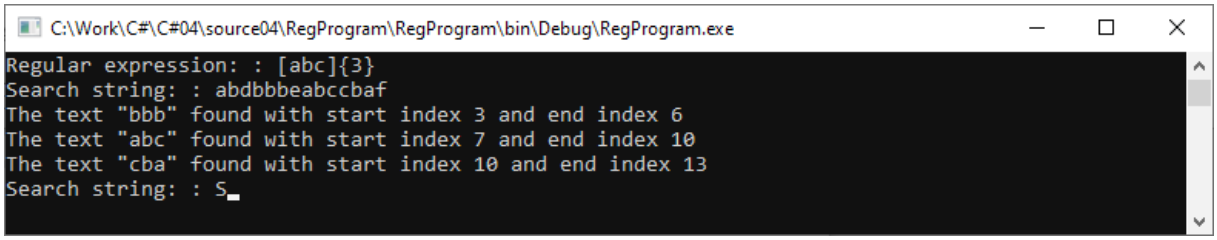
The above examples show that you often get a different result than expected. If, for example you want an expression that matches a certain number of characters, the syntax often will be something like the following:



If you want to search a group of characters you can use parentheses, where the following expression matches the sequence 123123:



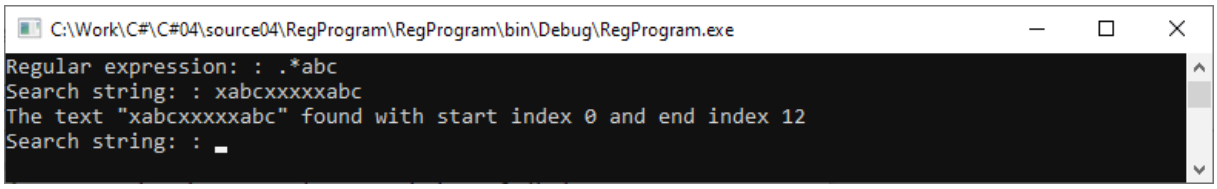
As another example, the following pattern matches 3 of the characters *a*, *b* and *c*:



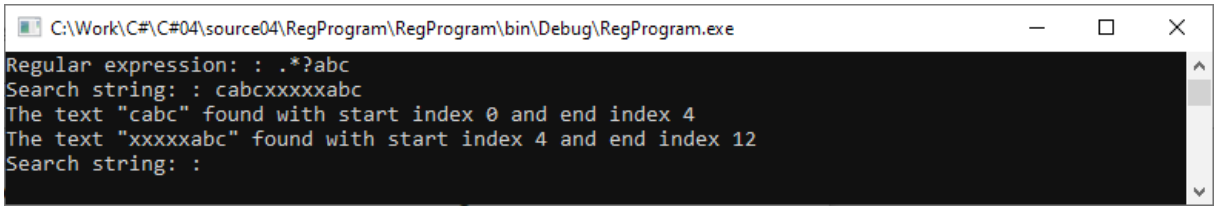
As shown in the table above there are three options for specifying quantifiers. As mentioned the search in the search-string moves the index until a match is found. If the quantifiers in the first column are used the index is moved as far as possible. A dot means the character class consisting of all characters, and the pattern

```
.*abc
```

therefore means 0 or more characters followed by abc. As the index moved as far to the right as possible the result of the following search is only in one match:



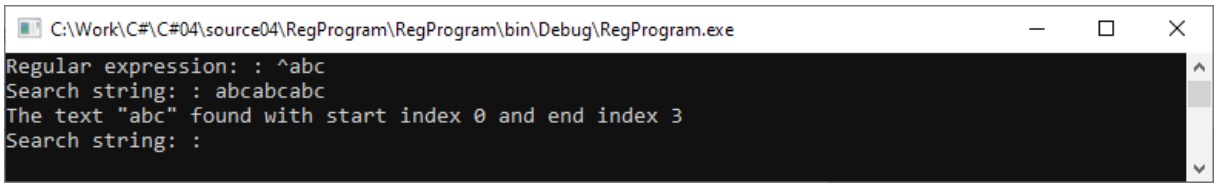
If the quantifiers in the second column are used, the search stops as soon as a match is found. Therefore results the following search two matches:

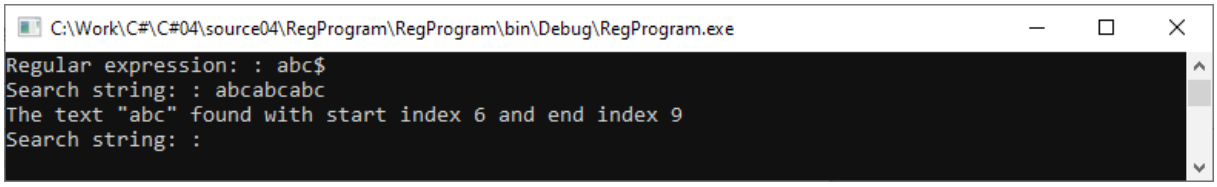


There are also some options to specify where in the search string a match must occur:

- ^ the start of the line
- $ the end of the line
- \b the start of a word
- \B not in the start of a word
- \A the start of the input
- \G the start of the previous match
- \z the end of the imput

Correspondingly, the following searches results in a single match:





The above examples demonstrate the basics regarding regular expressions, but there are several options, and the classes *Regex* and *Match* also has other useful methods. Here I refer to the documentation, and will instead end this introduction to regular expressions with a few examples.

In C#, the name of a variable start with a letter, and then there should follow any number of characters consisting of letters, digits and the character _. In addition, it is recommended that a variable name always starts with a lowercase letter. If you decides that it's the rules, a variable is defined by using the following regular expressions:

```
static void Test8()
{
  Regex regex = new Regex("^[a-z]+[a-zA-Z_0-9]*$");
  for (string str = Enter("? "); str.Length > 0; str = Enter("? "))
  {
    Match matc = regex.Match(str);
    Console.WriteLine(matc.Success);
  }
}
```

If you executes the method, you can enter strings, and the method will validate where the string is a valid variable. The pattern says that the string must start with at least one small letter:

```
[a-z]+
```

Next, the strings must end with any number of characters that are letters, digits or _:

```
[a-zA-Z_0-9]*$
```

As a complex example is shown a method using a regular expression to validate an email address. Note that I have already used the method in the previous book (the project *Students3*).

```
public static boolean isMail(String mail)
{
  return Regex.Match(mail,
    "^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@
      ((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\])|
      (([a-zA-Z0-9-]+.)+[a-zA-Z]{2,}))$"
    ).Success;
}
```

It is a very complex expression, and also it uses rules and operators that I not have highlighted above, so I will try to explain the expression. It starts with a character class:

```
[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+
```

It defines the small and capital letters, digits and a number of other special characters. You should note that when a special character is defined in a character class, it is unnecessary to escape them with a backslash. The exception is [,] and - because these characters are used to define the class. An email address consists of two parts separated by a @ character, as we call respectively the first name and the last name. The above character class defines thus what characters the first name may consist of, and that there must be at least one of these characters. You should note that the class allows many other characters than what is usual in mail addresses, but they are actually legal, at least if you includes mail addresses from all countries.

After the above character class follows the separator @, and the last name that is made up of two terms, so that a string should match the one or the other. You define this selection with the character |, and the first expression is therefore:

```
(\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\])
```

The expression matches a bracket begin, four integers of at least 1 and no more than 3 digits separated of a dot, and last a bracket final. This means that the expression matches an IP address in brackets.

The second expression has the form:

```
(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,})
```

Here is

```
[a-zA-Z0-9-]+
```

a character class that matches uppercase and lowercase letters, a hyphen and digits, and there must be at least one of these characters. Next, follow a dot. Of these groups must be at least one:

```
([a-zA-Z0-9-]+\\.)+
```

Finally there must be at least two letters. The result is that the last name is either an IP address or a server name. The entire expression is surrounded by the characters ^ and $, which means that there must be nothing in front or behind the expression.

EXERCISE 2: SOME REGULAR EXPRESSIONS

Create a console application program which you can call *RegProgram*. Add the class *Str* from exercise 1. You must then expand the class with the following methods:

```
/// <summary>
/// The method validates whether a string
    represents a binary number (a string
/// consisting of 0 and 1), when the string must start with 0 or 1.
/// </summary>
/// <param name="text">String which must
    represents a binary number</param>
/// <returns>True if text represents a binary number</returns>
public static bool IsBinary(this string text)
{
  throw new NotImplementedException();
}

/// <summary>
/// Validates where a string is hexadecimal
    integer without sign, when the syntax
/// should be as in C#, where the numbers starts with 0x.
/// </summary>
/// <param name="text">The string to be validated</param>
/// <returns>True, if text represents hexadecimal
    integer without sign</returns>
public static bool IsHex(this string text)
{
  throw new NotImplementedException();
}

/// <summary>
/// Validates whether a string is a long when
    the number must either be 0, or an
/// integer which may have a sign and must not start with 0, but otherwise
/// have maximum 17 digits.
/// </summary>
/// <param name="text">The string to be validated</param>
/// <returns>True, if text represents a long</returns>
public static bool IsLong(this string text)
{
  throw new NotImplementedException();
}
```

```
/// <summary>
/// Validates where a string is a double when
    the number may start with a sign and
/// there must be a decimal point followed by
    at least one digit. Moreover, it
/// should be possible to end the number with a eksponent part.
/// </summary>
/// <param name="text">The string to be validated</param>
/// <returns>True, if text represents a double</returns>
public static bool IsDouble(this string text)
{
  throw new NotImplementedException();
}
```

You should implements the methods using regular expressions.

You must also write e test program to test the methods.

3 DATE AND TIME

In many applications, data types are required for dates and times. C# has a type for that purpose called *DateTime*. It is a struct which is quite simple to apply, but there are still some things that you need to be aware of and actually dates and times are some of the things that complicate many programming tasks as different systems uses different representations and since there are also national differences.

The type *DateTime* represents dates and times within a range from January 1, year 1 00:00:00 (midnight) to December 31, year 9999 11:59:59. Time values are measured in 100 nanosecond units called ticks, and a particular date is the number of ticks after January 1, year 1 00:00:00 in the Gregorian calendar. The last is important as the type *DateTime* does not support the Julian calendar, and if you need to work on dates in the Julian calendar, special actions (= programming) are needed.

To use date and times in a program you for the most use objects of the type *DateTime*, and you can create an object in many ways:

```
using System;
using System.Globalization;

namespace DateProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(DateTime.Now);
      Console.WriteLine(new DateTime());
      Console.WriteLine(new DateTime(2020, 2, 9));
      Console.WriteLine(new DateTime(2020, 2, 9, 15, 45, 30));
      Console.WriteLine(DateTime.Today);
      Console.WriteLine(DateTime.Parse("2/9/2020 3:45:30 PM",
        CultureInfo.InvariantCulture));
      Console.WriteLine(DateTime.Parse("9/2/2020 15:45:30",
        CultureInfo.CurrentCulture));
      Console.WriteLine(DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"));
      Console.WriteLine(DateTime.Now.ToString("dd-MM-yyyy HH:mm:ss"));
    }
  }
}
```

The program shows some examples on how to create a *DateTime* object and I think that the examples are self-explanatory, but if you run the program the result could be:

The struct *DateTime* has many properties and methods that I do not want to show here, but you are encouraged to research the documentation so that you know which services the type provides. Most of it is there, but not all, and some of it I will return to in later examples.

PROBLEM 1: DATE METHODS

Create a new console application project which you can call *DateProgram*. Add a class called *Time*.

Sometimes in testing, it may be appropriate to measure how long it takes to execute a method (an algorithm). The project *StringProgram* from chapter 1 has a method called *GetTime()* which returns the number of milliseconds from start of the day to the current time. Add this method to the class *Time* and rename the method to *GetMilliseconds()*. Create another method *GetSeconds()* which do the same, but returns the value in seconds. Create also a method *GetMicroseconds()* that measure the time in microseconds. Test the three methods from your main program.

When you write programs where the user should enter a date or date and time you need a method to validate a *string* for a *DateTime* object. You must write such a method:

```
public static DateTime? From(string text)
```

where the method must return *null* if *text* does not represent a legal date. The method must accept some flexibility as to how the date is entered:

1. the date must start with values for year, moth and day separated by two characters that must be -, / or space
2. if year is the first value the last value must be day, and if year is the last field month must be the first field
3. a date must be followed by a time separated from the date with at least one space, and a time must have the format HH:MM:SS
4. in general the method should accept spaces where they do not lead to misunderstandings

Below is some examples on strings which the method must accept:

```
static void Test2()
{
  Console.WriteLine("::  " + Time.From("2020-02-09 12:3:45"));
  Console.WriteLine("::  " + Time.From("2020/2/9 12:3:45"));
  Console.WriteLine("::  " + Time.From("2020 2 9 12:3:45"));
  Console.WriteLine("::  " + Time.From(" 2020 - 2 - 9 12 : 3 : 45 "));
  Console.WriteLine("::  " + Time.From("2020-2-9"));
}
```

Also remember to test the method with illegal examples.

4 GENERIC TYPES

Generic types that have one or more parameters where the parameters are placeholders for types used in the generic type. Short it can be said that the aim is to write types that are general and can be used in many contexts, but such that the compiler can check that the types are used correct. Instead of the generic type one also refers to parameterized types corresponding to that it is types that depends on one or more parameters.

Both types and methods can be generic, and I will start with methods.

4.1 GENERIC METHODS

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 it 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 swaps two strings:

```
static void Test1()
{
  string s1 = "Svend";
  string s2 = "Knud";
  Swap(ref s1, ref s2);
  Console.WriteLine(s1);
  Console.WriteLine(s2);
}
```

As another example is below shown a generic method that prints an array:

```
static void Print<T>(params T[] arr)
{
  foreach (T t in arr) Console.Write(t + " ");
  Console.WriteLine();
}
```

a method that can be used to print any array:

```
static void Test2()
{
  int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  Print(primes);
  Print("Gorm", "Harald", "Svend", "Harald", "Knud");
}
```

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 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.

As an example I will should how to write a method which can sort an array of objects of any type. 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, and is the same as the type must implements the interface *IComparable* (see the book C# 1). The sorting method can then 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]);
  }
}
```

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, you should note the *where* part that expresses that the parameter type *T* must implement the interface *IComparable<T>* - and

then 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 *Swap()* method is used.

As the last example regarding generic methods I will return to the class *Cube* from C# 1, but this time with an extension so that it implements the interface *IComparable*:

```
class Cube : IComparable<Cube>
{
  private static Random rand = new Random();

  public int Eyes { get; private set; }

  public Cube()
  {
    Throw();
  }

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

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

  public int CompareTo(Cube c)
  {
    return Eyes < c.Eyes ? -1 : Eyes > c.Eyes ? 1 : 0;
  }
}
```

This means that the cubes can now be arranged such that two is less than a three, etc. Note that how they are ranked, is something that the programmer has specified in the implementation in the method *CompareTo()*, and that in principle one could have chosen any other arrangements.

As the next 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:

```
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 *Cube* 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<Cube>(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 compiler 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 compiler 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<Cube>(b);
```

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.

EXERCISE 3: SEARCH

When you have to search an element in an array a simple strategy is to start with the first element and compare it with the element to be searched, and then the second element, then the nest element and so on and continue until you find the element or reach the end of the array, and if so you know that the element is not there. This algorithm is called linear search and it is a widely used algorithm as it is often the only thing you can do. The disadvantage, of course, is that the algorithm can take a long time since at worst you have to compare with all elements in the array.

However, knowing that the array is sorted you can do it different and better. The idea is, that you start to compare the element to be searched for with the middle element in the array. If they are equals you are done. Else you test where the element to be searched is greater than the middle element, and if so you know that the element (if it is in the array) must be in the upper half of the array, but else it must be in the lower half. That way you have halved the number of elements to search. You then repeat the procedure but only on that half of the array where you know the element should be found. You continue until you find the element or the array to be searched is empty, and then you know the element is not in the array. The idea is, that you start to search in the full array, but for each iteration you search in an array which is only the half length of the previous array.

As an example look at the array:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

which is a sorted array with 15 elements. If you search for the element 38 the procedure is:

1. Start at index 7 and compare 19 with 38
2. As 38 is greater than 19 you must search among the elements from index 8 to index 14
3. The middle index in this half of the array is 11 and you compare 37 with 38
4. As 38 is greater than 37 you must search among the elements from index 12 to index 14
5. The middle index is then 13 and you must compare 43 with 38
6. As 38 is less than 43 you must search among the element from index 12 to index 12
7. The middle index is 12 and you must compare 41 with 38
8. Next time you get an empty interval and you are done, and 38 is not found

This algorithm is called binary search and is extremely effective but it is of course important to note that a prerequisite is that the array is sorted.

You must create a console application project that you can call *SearchProgram*. You must write two methods that implements respectively linear search and binary search when both methods must be written as generic methods. Both methods must return a value that is an *int* and is the index for the first position in the array, where the element is found. If the element is not found the methods must return -1.

When you have written the methods you must test them to be sure that they work as they should.

You should then and a test method

```
static void Test2(int n)
{
}
```

that performs the following:

1. create an *int* array with *n* elements
2. fill the array med the even numbers from 0 onwards, the result is a sorted array
3. initialize a random number between 0 and *n / 2*
4. search the array for the number *2n* using linear search and record the number of milliseconds for the operation
5. search the array for the number *2n* using binary search and record the number of milliseconds for the operation

6. search the array for the number *2n+1* using linear search and record the number of milliseconds for the operation
7. search the array for the number *2n+1* using binary search and record the number of milliseconds for the operation
8. print the results, the number of milliseconds for the search operations

Can you observe a difference between linear search and binary search?

4.2 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 Pair<T1, T2>
{
  public T1 Arg1 { get; set; }
  public T2 Arg2 { get; set; }

  public Pair()
  {
  }

  public Pair(T1 t1, T2 t2)
  {
    Arg1 = t1;
    Arg2 = t2;
  }

  public void Clear()
  {
    Arg1 = default(T1);
    Arg2 = default(T2);
  }

  public override string ToString()
  {
    return string.Format("({0}, {1})", Arg1.
    ToString(), Arg2.ToString());
  }
}
```

It is a very 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()*. The class also has a method *Clear()* with the only purpose to show the syntax that you can assign a property a value using the keyword *default*. The class *Pair* does not know what the parameter types *T1* and *T2* means and can then not assign values to properties of these types, but all types has a default value which for reference types is *null* while for simple types as numeric types is 0. Using the keyword *default* a generic class can assign default values to properties / variables.

You should primarily note the syntax of a parameterized type, and that there may be one or more type parameters (this also applies to a parameterized method).

Below is a method that uses the type:

```
static void Test5()
{
  Pair<int, int> p1 = new Pair<int, int>(2, 3);
  Pair<int, double> p2 = new Pair<int, double>();
  p2.Arg1 = 23;
  p2.Arg2 = Math.PI;
  Pair<string, Cube> p3 = new Pair<string, Cube>("Red", new Cube());
  Console.WriteLine(p1);
  Console.WriteLine(p2);
  Console.WriteLine(p3);
}
```

There is not much to explain and if the method is executed, the result is:

4.3 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.

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);
}
```

Note, that the interface is generic, and hence that an interface in the same manner as a class can be defined generic. 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; }
  }
```

```
  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;
  }

  private void Expand()
  {
    T[] temp = new T[2 * elems.Length];
    for (int i = 0; i < count; ++i) temp[i] = elems[i];
    elems = temp;
  }
}
```

There is not much to explain about the code, 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 for 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 is 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, a principle also used by the collection classes.

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]);
```

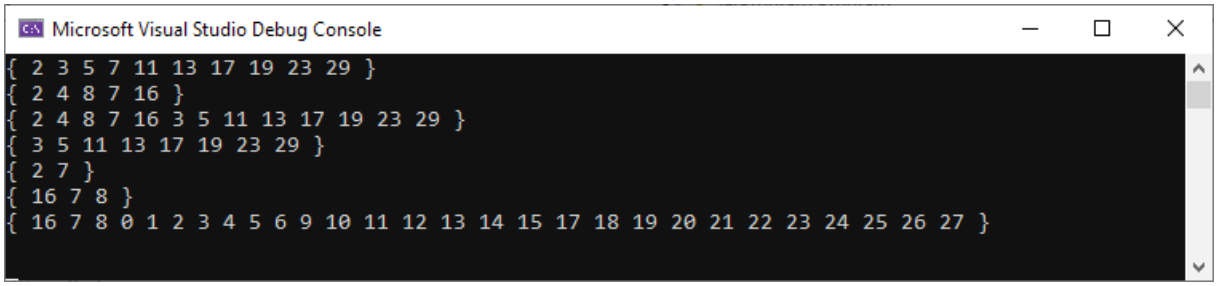
It consists of a loop, which runs through the sets elements. In principle it is fine, but for each element the statement called *Contains()* which calls *IndexOf()*, which then performs a second loop, and it gives a bad performance. It is said that the algorithm has a poor time complexity.

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.

As explained above, the class *Set<T>* is implemented so that its methods are not quite as effective as we would like. 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*.

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);
}
```



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 a Set.

Below is a simple application of the struct *Point*:

```
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.

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);
}
```

I mentioned above, that the class is not very efficient. To examine the effectiveness, I would try the following method:

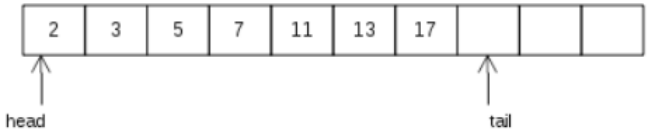
```
static void Test4()
{
  int N = 10000;
  ISet<int> A = Create(N, 2 * N);
  ISet<int> B = Create(N, 2 * N);
  Stopwatch sw = new Stopwatch();
  sw.Start();
  ISet<int> C = A.Union(B);
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds);
  sw.Start();
  ISet<int> D = A.Intersection(B);
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds);
}

static ISet<int> Create(int n, int m)
{
  ISet<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 uses 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 (each operation takes between 3 to 4 seconds on my machine):

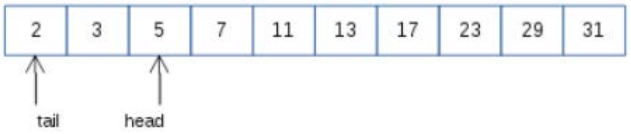
PROBLEM 2: A BUFFER

You must write a generic class representing a circular buffer. A buffer is a container (a collection), which may contain objects, and to the buffer has associated methods that manipulates the content. A buffer can be implemented in several ways, but here you should look at a so-called circular buffer, a buffer with room for a certain number of objects. Such a buffer can be easily implemented by means of an array:

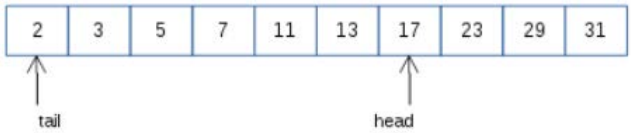
Basically, there are to a buffer associated two operations called *Insert()* and *Remove()*, where the first inserts an element into the buffer, while the other removes the oldest element, the element that has been longest in the buffer. The figure above illustrates an empty buffer, and the arrow *tail* is the index where the next element should be inserted. Similarly, *head* is the index of the next element to be removed. Initially, the buffer is empty and the arrows points to the same position. Below is how the buffer looks after inserted 7 elements, the method *Insert()* is executed 7 times:



If you then remove two elements, the method *Remove()* is executed 2 times, and then add 3 elements, you get the result:



Here you should especially note that the index *tail* wraps around so that the next element to be inserted is at position 0. The next figure shows the buffer, after 4 additional elements are removed:



and the latter figure shows the result after insertion of two additional elements:

The name *circular buffer* is derived from the indexes that wraps around when they reach the end of the array.

Create a console application project and add the following interface which defines a buffer:

```
public interface IBuffer<T>
{
  /// <summary>
  /// The number of elements in the buffer.
  /// </summary>
  int Count { get; }

  /// <summary>
  /// True if the buffer is empty.
  /// </summary>
  bool IsEmpty { get; }

  /// <summary>
  /// True if the buffer is full.
  /// </summary>
  bool IsFull { get; }

  /// <summary>
  /// Returns the oldest (the first) element in
      the buffer without removing it
  /// if the buffer is not empty.
  /// </summary>
  /// <returns>The oldest (the first) element of the buffer</returns>
  T Peek();

  /// <summary>
  /// Returns the oldest (the first) element in
      the buffer and remove the element
  /// from the buffer, if the buffer is not empty.
  /// </summary>
  /// <returns>The oldest (the first) element of the buffer</returns>
  T Remove();

  /// <summary>
  /// Adds an element to the buffer if the buffer is nou full.
  /// </summary>
  /// <param name="elem">The element to be added</param>
  void Insert(T elem);
}
```

You must then add a class *Buffer* that must implements the above interface as a circular buffer. When you have written the class you can test it with the following program:

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

    static void Main(string[] args)
    {
      IBuffer<int> buffer = new Buffer<int>(5);
      for (int i = 0; i < 100; ++i)
        if (rand.Next(2) == 0)
          try
          {
            Console.WriteLine("<< " + buffer.Remove());
          }
          catch (Exception ex)
          {
            Console.WriteLine(ex.Message);
          }
        else
          try
          {
            int t = rand.Next(90) + 10;
            buffer.Insert(t);
            Console.WriteLine(">> " + t);
          }
          catch (Exception ex)
          {
            Console.WriteLine(ex.Message);
          }
    }
  }
}
```

The program creates a *Buffer* with room for five numbers. The program iterates over a loop where it randomly either insert a number or remove a number. Some operations will fail because the buffer is either full or empty.

5 COLLECTION CLASSES

A collection class is a container for objects, and a good example is the type *List*, which is discussed and used many times in the previous books. A collection class is a little more than just a container for objects since the class also provides a number of methods available, which are used to manipulate the container's objects. C# has other collection classes than *List*, and they differ in terms of how they are implemented and store their objects, as well as the services they provides in the form of methods. Overall reflects these collection classes the tasks that typically occurs in programs to manipulate a family of objects.

It should immediately be said that the goal of this chapter is to give an overview of the collection classes, which classes exist and what they can be used for, but it is not a detailed review of the collection classes and how they work. However, it is the subject for later books in this series.

C#'s collection classes are found in the namespace *System.Collections.Generic* which contains generic interfaces and classes, and the description of the types is the goal of the following chapter.

The basic interfaces are:

- *IComparer<T>* which defines how to compare objects
- *IEnumerable<T>* which defines that a collection is enumerable
- *IEnumerator<T>* which defines an enumerator
- *ICollection<T>* which defines all the basic characteristics for collections
- *IList<T>* which defines a list
- *ISet<T>* which defines a set
- *IDictionary<K, T>* that defines a collection of key / value pairs

In this chapter I will mention the following 9 concrete collection classes, where the list shows which interfaces they implements:

- *List<T>* *ICollection<T>, IList<T>, IEnumerable<T>*
- *LinkedList<T>* *ICollection<T>, IEnumerable<T>*
- *Queue<T>* *ICollection<T>, IEnumerable<T>*
- *Stack<T>* *ICollection<T>*
- *Dictionary<K, T>* *ICollection<T>, IDictionary<K, T>, IEnumerable<T>*
- *SortedDictionary<K, T>* *ICollection<T>, IDictionary<K, T>, IEnumerable<T>*
- *SortedList<K, T>* *ICollection<T>, IDictionary<K, T>, IEnumerable<T>*
- *SortedSet<T>* *ICollection<T>, ISet<T>, IEnumerable<T>*
- *HashSet<T>* *ICollection<T>, ISet<T>, IEnumerable<T>*

The program *CollectionProgram* shows through examples how to use the 9 classes.

5.1 LIST<T>

This type is described in the book C# 1 and can best be interpreted as a dynamic array. That is a type that is used in many ways as an array, but which can grow dynamically as needed. This is in contrast to a regular array, where you have to specify how many elements will be available when creating the array. However, one should not go 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 thus a structure that has a capacity at a given time, and then elements can be added to the end of the list. The picture could thus be as shown below, where the capacity is 10 while using the first 7 places:

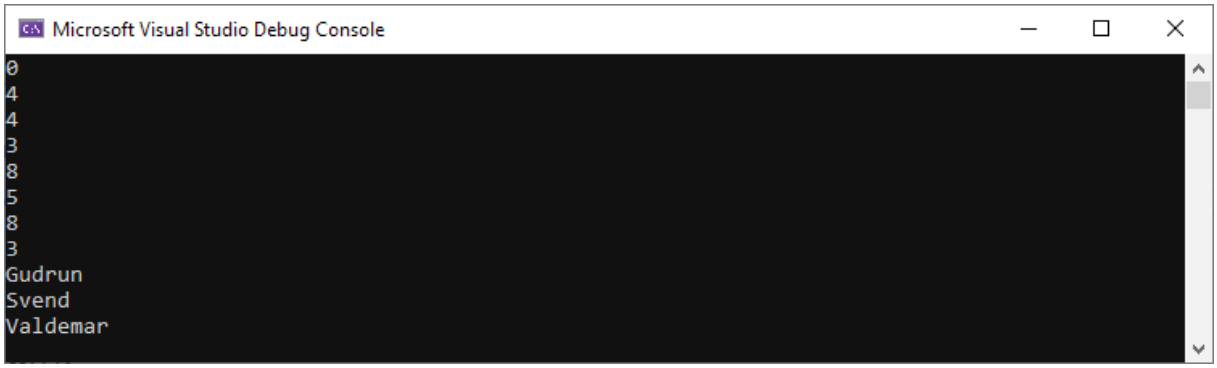
You can add elements where the arrow points, and if you exceed the capacity at some point, it will automatically be expanded. This is done by doubling the capacity.

In addition to adding items to the end of a List, the class offers a variety of other methods, for example that you can delete an element, insert an element in a specific position, etc. Internally, a *List* is an array, and this also means that you have to be aware of the complexity of the different methods. A *List* cannot have empty places, which means that, for example, if an item is deleted, all items to the right of the deleted item should be moved to the left. Similarly, if you insert an element. Then all elements to the right of the place where the element is inserted must be moved one position to the right to make room for the new element. In contrast, *Add()* which adds an element to the end of the list, is extremely effective as it can immediately place the element where the arrow points, except for the situation where it is necessary to double the capacity.

As an example the following method creates a list to strings:

```
static void Test1()
{
  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);
  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]);
}
```

First, the list's capacity is initially printed, which is 0. However, it is possible to create a list where, as a parameter to the constructor, you specify the starting capacity. After an element is added, the capacity is printed again. It is now 4, which means that the first time an item is added to the list, space for 4 items is allocated. After two more elements have been added, the capacity is reprinted as well as the number of elements, respectively 4 and 3, which is not surprising. As the next step, two elements are inserted at the beginning of the list (in position 0) using the method *Insert()*. The capacity is then 8 (has doubled) and the number of elements is 5. Then two elements are deleted, after which the capacity is still 8. It is worth noting that a *List* does not automatically shrink. The last for loop prints the list items, and the important thing here is to note that you can use the index operator.



The collection class *List* is the most used collection class.

EXERCISE 4: A LIST

Create a console application project which you can call *ListProgram*. The program must create a *List<int>*. The program must add 10000000 random numbers between 0 and 100 to the list and then calculates the sum of the numbers in the list. The program must print:

1. The number of milliseconds to add the 10000000 integers to the list
2. The number of milliseconds to calculate then sum
3. The number of elements in the list
4. The capacity of the list
5. The sum

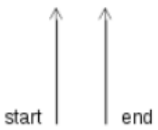
Remember, that you can measure the time using a *StopWatch* object.

5.2 LINKEDLIST<T>

It's also a list, but it is implemented in a whole different way, as a double linked list. If you creates a *LinkedList*

```
LinkedList<int> list = new LinkedList<int>();
```

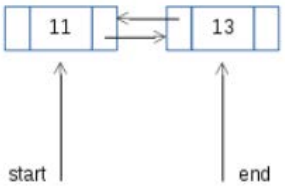
you have an empty list, which you should think about in the following way:



that is two pointers pointing respectively to the start and end of the list. That the list is empty means that the two pointers both are *null*. The class has a method *AddLast()* that works the same way as the method *Add()* in a *List* and adds an element to the end of the list. If you perform the following statements

```
list.AddLast(11);
list.AddLast(13);
```

the picture of the list is as below:

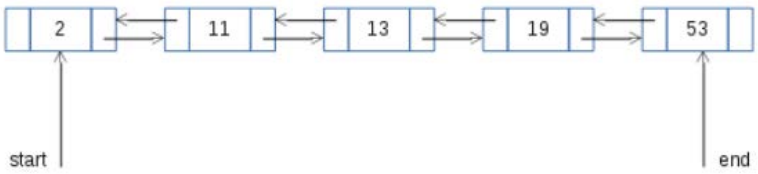


A *LinkedList* consists of so-called *nodes*, where a node includes a data item and a pointer to the previous node, and a pointer to the next node in the list. The first node (data element 11) has no predecessor, and its pointer to the previous node is *null*, and, similarly, the pointer to the next node in the node with the data element 13 is also *null*. In turn, node 11 points forward on node 13, while node 13 points back to node 11.

Compared to a *List* a *LinkedList* has several methods to add items to the list:

```
list.AddFirst(2);
list.AddLast(53);
list.AddAfter(list.Find(13), 19);
```

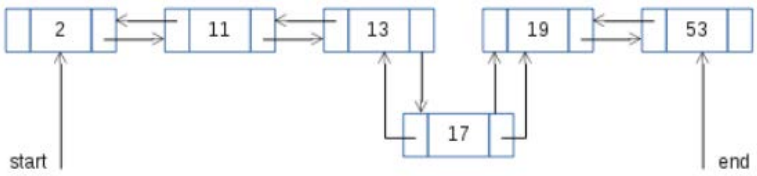
Here *AddFirst()* adds and element at the start of the list. *Find()* is a method that finds the node in the list which contains the argument, and in this case it is 13. *Find()* returns an element of the type *LinkedListNode<int>* and the node is used in *AddAfter()* to insert a new node containing 19 after that node:



The idea with a *LinkedList* is that, when compared with a *List* it is simple to insert an element and delete an element in the middle of the list. If, for example you performs the statement

```
list.AddAfter(list.Find(13), 17);
```

element 17 has to be inserted between 13 and 19:

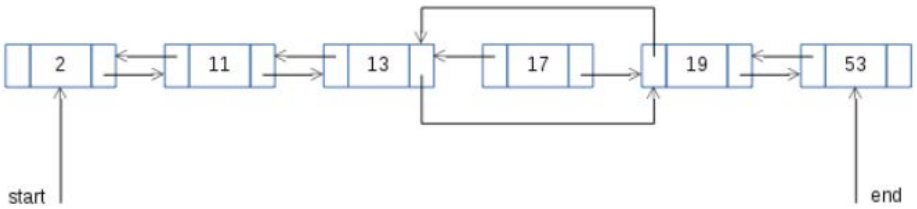


This means to create a new node, and then change the 4 pointers, but you should note, that you not in the same way as with a *List* have to move all elements to the right of the place where the new element should be inserted, but note that you have to find the place before with the method *Find()*.

If you assume that the element 17 is inserted and you performs the statement

```
list.Remove(list.Find(17));
```

is the result



This means that there are changed two pointers, and there is thus no longer references to the item 17, which are then removed by the garbage collector. Again, the benefit is the same that deletion requires few operations. Both insertion and deletion of elements is simple, but requires special implementations for both inserting and deleting of elements in the ends of the list.

The following method creates a *LinkedList<string>* and add some names to the list:

```
static void Test03()
{
  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(“-----------------”);
}
```

There are four ways to inserts an element in a linked list

1. add an element to the end of the list
2. add an element as the first element in the list
3. add an element after an existing element
4. add an element before an existing element

Regarding the last, note how to use the method *Find()* to find the element to be inserted relative to. It returns a *LinkedListNode*, which is a structure similar to a node.

Note the three *Print()* methods. About the first is not much to say, as it prints the list with a usual *foreach* loop. In the second method, a property is used to get a reference to the last node, again note the type *LinkedListNode*. Next, the list is iterated from behind. In a *LinkedList* you cannot reference the elements with an index, but there is a method *ElementAt()* that produces the same result. It shows the last print method. *ElementAt()* is not a method in the class *LinkedList*, but is an extension method defined in a LINQ class (note the using statement). Note also that *ElementAt()* does in fact every time count from the beginning of the list onwards, so it is not an appropriate solution to the print problem.

Whether to use a *LinkedList* or a *List* is determined by the task, as both data structures have their advantages and disadvantages. However, there is no doubt that a *List* is the most frequently used.

EXERCISE 5: A LINKEDLIST

You must write a program which do the same as the program from exercise 4, when

1. the list must this time be a *LinkedList<int>*
2. you should add numbers with *AddLast()*
3. the statement which prints the capacity must be removed

When you run the program I think you will observe that this program is a bit slower than the previous program.

Change the program such that you fill the list with the numbers 1, 2, 3, 4, 5, ... and using the method *AddFirst()*. Test the program again. I do not think you can observe any difference. This means that the sum should be

5000050000, n = 100000
500000500000, n = 1000000
50000005000000, n = 10000000

Try to fill the list as follows, where n is the number of integers that should be added to the list:

```
Add 1 to the list
Loop from 2 to n and for each iteration:
  Use the method Find() to find a random integer in the list
  Select random AddBefore() or AddAfter() and
  add the loop index to the list
  using the selected method
```

Test the program again. I think you must lower 10000000, maybe 10000 or 100000.

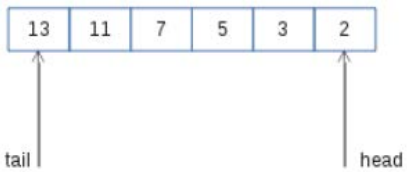
Change the algorithm:

```
Add 1 to the list
Let node be a reference to node containing 1
Loop from 2 to n and for each iteration:
  Select random AddBefore() or AddAfter() and
  add the loop index to the list
  using the selected method and with node as the first argument
```

Test the program again. Conclusion: The method *Find()* is not free.

5.3 QUEUE<T>

A queue is a data structure where you primary can add an item and removing an item but such that the element that is removed, always is the oldest element, it's the element been the longest in the queue. Typically, one have the following picture of a queue



which adds elements where tail is pointing and remove the element where head is pointing. The class *Queue<T>* is a very simple class, and the two fundamental methods are

1. *Enqueue()* to add an element to the queue
2. *Dequeue()* to remove an element from the queue

As an example the following method creates a queue, insert 10 elements in the queue and the prints the content of the queue with a *foreach* loop. The the method add 990 other elements, and as the last the method remove all elements from the queue while calculating the sum:

```
static void Test04()
{
  Queue<int> q = new Queue<int>();
  for (int i = 1; i <= 10; ++i) q.Enqueue(i);
  foreach (int t in q) Console.Write(t + " ");
  Console.WriteLine();
  for (int i = 11; i <= 1000; ++i) q.Enqueue(i);
  long s = 0;
  while (q.Count > 0) s += q.Dequeue();
  Console.WriteLine(s);
}
```

The class *Queue<T>* is implemented as a circular queue, but if you add an element, and the queue not has room for the element the capacity is expanded in the same way as explained for a *List*.

The class *LinkedList* has methods *AddFirst()* and *AddLast()* and also methods *RemoveFirst()* and *RemoveLast()* that are all very effecitive. If you use a *LinkedList* with the methods *AddFirst()* and *RemoveLast()* it is actual a queue.

EXERCISE 6: A QUEUE

Create a console application project as you can call *QueueProgram*. Add a method

```
static long Test1(int n)
{
}
```

The method must create a *Queue<int>* and add the numbers

1, 2, 3, 4, ..., n

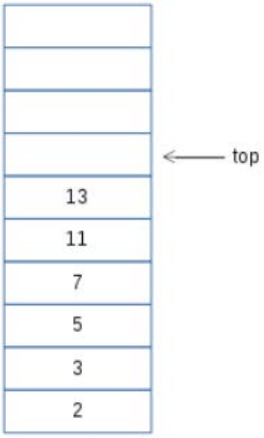
to the queue. The method should then remove all numbers from the queue and calculates and return the sum.

In *Main()* you should call the method and determine how long time it takes to executes the method. Print the time and the sum.

Write a corresponding method *Test2()* which performs the same, but instead than a *Queue<int>* using a *LinkedList<int>*. Test also this method from *Main()* and note where you can observe a difference in time.

5.4 STACK<T>

A stack is look likes a queue, and it is again a data structure, where you can add and remove elements, but this time it is the element that was last added to the stack, that is is removed. A stack can be illustrated as shown here:



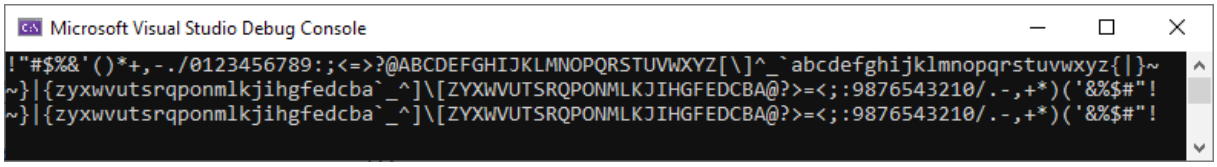
where there are 6 elements on the stack. The arrow indicates the place where the next element should be added. A stack is called a LIFO structure for *Last In First Out*, while a queue is called a FIFO structure for *First In First Out*, and it is exact the difference between the two collection classes and their use. The two fundamental methods in the class *Stack<T>* is

1. *Push()* which add an element at the top of the stack
2. *Pop()* which remove the element at the top of the stack

As an example the following method push the characters with code from 33 to 126 on a stack, iterates over the content of the stack, and the remove all elements from the stack:

```
static void Test5()
{
  Stack<char> stack = new Stack<char>();
  for (int i = 33; i < 127; ++i)
  {
    char c = (char)i;
    Console.Write(c);
    stack.Push(c);
  }
  Console.WriteLine();
  foreach (char c in stack) Console.Write(c);
  Console.WriteLine();
  while (stack.Count > 0) Console.Write(stack.Pop());
  Console.WriteLine();
}
```

You should note (see below) that when you iterates a stack in a *foreach* loop it happens from top to bottom. Also note, that when you pop the stack you see the elements in the opposite order as they are added to the stack.



EXERCISE 7: A STACK

You should write a program called *StackProgram* that should be used to sort an array. It is possible to sort an array using two stacks, and the following algorithm can be used:

```
for each element t i in the array repeat
{
  as long t i less than the top of the left stack do
  {
    pop the left stack and push the element on the right stack
  }
  as long t is greater than the top of the right stack do
  {
    pop the right stack and push the element on the left stack
  }
  push t on the left stack
}
as long the left stack is not empty do
{
  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
}
```

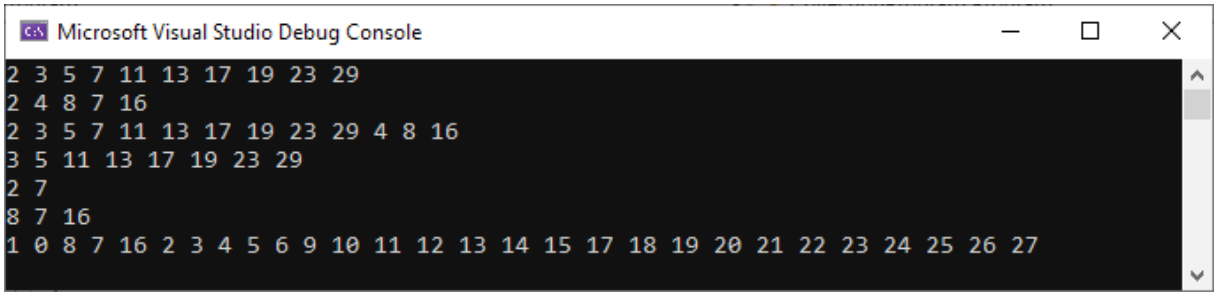
Write a generic method *StackSort(T[] arr)* that implements the above algorithm and sorts an array.

Write a *Main()* method to test the sort method.

5.5 HASHSET<T>

The interface *ISet* defines a mathematical set and including the operations that you would typically expect to perform on sets. In the previous chapter I write a class that could represent a set using a *List*, but I also noticed that the implementation of the set was not especially effective. C#, however, has a class *HashSet* which also represents a set, and is in turn effective. Consider the following method, which is almost the same method I used in the previous chapter:

```
static void Test06()
{
  ISet<int> A = new HashSet<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 HashSet<int>();
  B.Add(2);
  B.Add(4);
  B.Add(8);
  B.Add(7);
  B.Add(16);
  Print(A);
  Print(B);
  ISet<int> C = new HashSet<int>(A);
  C.UnionWith(B);
  ISet<int> D = new HashSet<int>(A);
  D.ExceptWith(B);
  ISet<int> E = new HashSet<int>(A);
  E.IntersectWith(B);
  Print(C);
  Print(D);
  Print(E);
  B.Remove(2);
  B.Remove(4);
  B.Remove(6);
  Print(B);
  for (int i = 0; i < 28; ++i) B.Add(i);
  Print(B);
}

static void Print<T>(ISet<T> S)
{
  foreach (T t in S) Console.Write(t + " ");
  Console.WriteLine();
}
```



The only difference is that the sets objects are defined in another way and as objects of the type *HashSet<int>* and the method for union, intersection and difference are called something else. They also works a bit differently as for example

```
C.UnionWith(B);
```

that creates the union of *C* and *B* but such that *C* is changed and contains the result. When I do not want to change *A* it is necessary to create C as a copy of *A* before creating the union.

A *HashSet* is implemented by hashing, which is a technique in which an element in a collection can be found by a calculation. Internal, the collection is an array of sufficient size and the individual objects are placed in the array. When you add a new object, the class calculates using an algorithm where in the array the object must be placed, and it is placed there (if the place is not already used). Similarly, if you want to get a specific object in the collection, then the class just calculate the object's place and test whether it exists. This means that there is direct access to the individual objects without any search, and that is exactly what makes the data structure extremely effective.

An *Object* has as mentioned a method called *GetHashCode()*, which returns an *int*. When objects of a given type can be stored in a *HashSet*, the object's class must overrides the method *GetHashCode()* with value semantic. It is a requirement that if two objects that are *Equals()* they also must have the same hash code, but it is also the only formal requirements. On the other hand, it is the programmer's responsibility to ensure that *GetHashCode()* is implemented in such a way that the method returns values that are uniformly distributed. It should be particularly noted that it is not a requirement that the two objects with the same hash code are equals. To store objects in a *HashSet* you must then ensure that the objects has a type which overrides both *Equals()* and *GetHashCode()*.

In the previous chapter I showed a generic class *Pair<T1, T2>* and in the following examples I want to store objects of that type in a *HashSet*. To do that it is necessary to expand the class and implements the methods *Equals()* and *GetHashCode()*:

```
class Pair<T1, T2>
{
  public T1 Arg1 { get; set; }
  public T2 Arg2 { get; set; }

  public Pair()
  {
  }

  public Pair(T1 t1, T2 t2)
  {
    Arg1 = t1;
    Arg2 = t2;
  }

  public void Clear()
  {
    Arg1 = default(T1);
    Arg2 = default(T2);
  }

  public override bool Equals(object obj)
  {
    if ((obj == null) || !this.GetType().
    Equals(obj.GetType())) return false;
    Pair<T1, T2> pair = (Pair<T1, T2>)obj;
    return Arg1.Equals(pair.Arg1) && Arg2.Equals(pair.Arg2);
  }

  public override int GetHashCode()
  {
  return Arg1.GetHashCode() ^ Arg2.GetHashCode();
  }

  public override string ToString()
  {
    return string.Format("({0}, {1})", Arg1.ToString(), Arg2.ToString());
  }
}
```

Accept the implementation of *GetHashCode()* as it is, but it is a XOR of the hash codes for two properties. As an example the following method adds 10 *Pair<int, string>* objects to a *HashSet*:

```
static void Test07()
{
  ISet<Pair<int, string>> zipcodes = new HashSet<Pair<int, string>>();
  zipcodes.Add(new Pair<int, string>(7800, "Skive"));
  zipcodes.Add(new Pair<int, string>(7900, "Nykøbing M"));
  zipcodes.Add(new Pair<int, string>(8800, "Viborg"));
  zipcodes.Add(new Pair<int, string>(7700, "Thisted"));
  zipcodes.Add(new Pair<int, string>(7620, "Lemvig"));
  zipcodes.Add(new Pair<int, string>(7600, "Struer"));
  zipcodes.Add(new Pair<int, string>(7500, "Hostebro"));
  zipcodes.Add(new Pair<int, string>(7400, "Herning"));
  zipcodes.Add(new Pair<int, string>(7950, "Erslev"));
  zipcodes.Add(new Pair<int, string>(7800, "Skive"));
  foreach (Pair<int, string> zipcode in
  zipcodes) Console.WriteLine(zipcode);
}
```

Except it works you must note two things:

1. The result only show 9 objects. The method add the same object two times, and if you add an object to a *HashSet* and the set already contains the object the operation is just ignored.
2. It seams like the objects are stored in the order they are added. You cannot be sure of this, and with a *HashSet* you must never assume anything about the physical location of the objects in the structure.

A *HashSet* is a very effective data structure. If you execute the following method which is the same method that I used in the previous chapter (using my own set class):

```
static void Test08()
{
  int N = 10000000;
  ISet<int> A = Create(N, 2 * N);
  ISet<int> B = Create(N, 2 * N);
  Stopwatch sw = new Stopwatch();
  sw.Start();
  ISet<int> C = new HashSet<int>(A);
  C.UnionWith(B);
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds);
  sw.Start();
  ISet<int> D = new HashSet<int>(A);
  D.IntersectWith(B);
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds);
}
```

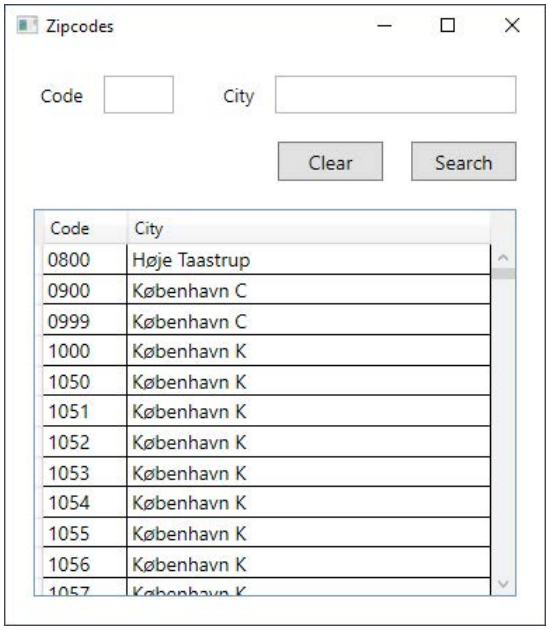
you get pretty much the same result as in the previous chapter, but this time there are 10000000 elements instead of 10000.

5.6 SORTEDSET<T>

A *SortedSeta<T>* is a set where the elements are sorted. Else the class has the same properties and methods like the class *HashSet<T>*. Of course the parameter type must implements *IComparable<T>*, but else you use the type exactly the same way as *HashSet<T>*. The difference is that it costs to require the elements to be sorted, so you should not use a *SortedSet* rather than a *HashSet* unless you have a reason. However, it is important to note that a *SortedSet* is by no means ineffective.

PROBLEM 3: A ZIP CODE TABLE

You must write a program to search in Danish zip codes:



If you enter text in the fields *Code* and *City* and click on *Search* the program should find all zip codes where the code starts with the entered value for code and the city name contains the entered value for city. The last button should be used to clear the two fields.

Create a WPF application project. The source directory for this book contains a file *zipcodes* which is a text file that contains the zip codes with one line for each zip code where the code and city name are separated by a semicolon. Copy the file to the folder with the program file.

Add a class to the project which represents a zip code:

```
class Zipcode : IComparable<Zipcode>
{
  public string Code { get; private set; }
  public string City { get; private set; }
```

Then add another class where in the constructor to read the file and initialize a *SortedSet<Zipcode>* with *Zipcode* objects.

The user interface should have a *DataGrid*, and you should define the component as:

```
<DataGrid x:Name="dataGrid" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Code" FontSize="14" Width="60"
      Binding="{Binding Code}" />
    <DataGridTextColumn Header="City" FontSize="14" Width="*"
      Binding="{Binding City}" />
  </DataGrid.Columns>
</DataGrid>
```

Then you can update the component to assign the property *ItemsSource* a new value.

When you have written the program and it all seams to work you should change the collection with *Zipcode* objects from a *SortedSet<Zicpcode>* to a *HashSet<Zipcode>*.

5.7 DICTIONARY<K,T>

A dictionary is a collection where the individual elements are identified by a key. Therefore, a dictionary is sometimes referred to as a collection of key / value pairs, each value being identified by a key. You can compare a dictionary with a list, because in a list the elements are identified by an index. When you find an item in the list, you do not search for the item, but you calculate its place based on the index. Similarly, with a dictionary, when you need to find an element, you calculate the element's place based on the value of the key, the collection of keys is a *HashSet*, and assigned with each key is a reference that tells where the value is stored.

The type is a generic type parametrized by two parameters, where the first is the key, while the other is type for the values. Both types may be everything.

Below is a method that add 6 objects to a dictionary. Both key and value has the type *string*:

```
static void Test09()
{
  IDictionary<string, string> map = new Dictionary<string, string>();
  map.Add("Knud", "King");
  map.Add("Gudrun", "Witch");
  map.Add("Svend", "Warrior");
  map.Add("Olga", "Soothsayer");
  map.Add("Valdemar", "Executioner");
  map.Add("Abelone", "Wise wife");
  Print1(map);
  map["Gudrun"] = "Her husband's wife";
  Print1(map);
  Print2(map);
  Print3(map);
}

static void Print1(IDictionary<string, string> map)
{
  for (int i = 0; i < map.Count; ++i) Console.
  WriteLine(map.ElementAt(i));
  Console.WriteLine("-----------------");
}

static void Print2(IDictionary<string, string> map)
{
  foreach (string name in map.Keys) Console.WriteLine(map[name]);
  Console.WriteLine("-----------------");
}

static void Print3(IDictionary<string, string> map)
{
  foreach (string job in map.Values) Console.WriteLine(job);
  Console.WriteLine("-----------------");
}
```

First, note how to add values to a dictionary. Then note the first print method, where you print 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 by using the key as an index:

```
map["Gudrun"] = "Her husband's wife";
```

Finally, note the last two print methods, where the first runs over all keys and the second over all values. Here you should note how to get a collection with all keys (the method *Print2()*) and a collection with all values (the method *Print3()*).

I will change the above method and use the following type as key:

```
class Name
{
  public string Firstname { get; set; }
  public string Lastname { get; set; }

  public Name(string firstname, string lastname)
  {
    Firstname = firstname;
    Lastname = lastname;
  }

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

and the method could be written as:

```
static void Test10()
{
  IDictionary<Name, string> map = new Dictionary<Name, string>();
  map.Add(new Name("Knud", "Madsen"), "King");
  map.Add(new Name("Gudrun", "Jensen"), "Witch");
  map.Add(new Name("Svend", "Andersen"), "Warrior");
  map.Add(new Name("Olga", "Olesen"), "Soothsayer");
  map.Add(new Name("Gudrun", "Jensen"), "Her husband's wife");
  Print(map);
}

static void Print(IDictionary<Name, string> map)
{
  foreach (Name name in map.Keys) Console.WriteLine(map[name]);
  Console.WriteLine("-----------------");
}
```

Note that it can be translated and run, but the result is not as expected, as key *Gudrun Jensen* appears twice. The reason is that the type *Name* does not override *Equals()* and *GetHashCode()*. As mentioned, a dictionary is based on a hash function to calculate the place of the individual elements, which in turn uses the method *GetHashCode()*. Therefore, the key type must always override *Equals()* and *GetHashCode()*:

```
class Name
{
  public string Firstname { get; set; }
  public string Lastname { get; set; }

  public Name(string firstname, string lastname)
  {
    Firstname = firstname;
    Lastname = lastname;
  }

  public override bool Equals(object obj)
  {
    if ((obj == null) || !this.GetType().
    Equals(obj.GetType())) return false;
    Name name = (Name)obj;
    return Firstname.Equals(name.Firstname) &&
    Lastname.Equals(name.Lastname);
  }

  public override int GetHashCode()
  {
    return Firstname.GetHashCode() ^ Lastname.GetHashCode();
  }

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

If you now performs the method you get an exception, as you now try to add two objects with the same key, but if you remove one of them, it all runs as it should.

5.8 SORTEDDICTIONARY<K, T>

This class is in the same way as *Dictionary<K, T>* an *IDictionary<K, T>* and thus a collection of key / values pairs with the difference that the keys are sorted. In principle, it is a question of having the keys stored in *SortedSet* instead of a *HashSet*, and compared to a *Dictionary*, a *SortedDictionary* has the same advantages and disadvantages that apply to a *SortedSet* or a *HashSet*. Thus, you should only select *SortedDictionary* over a *Dictionary* if you need the keys to be sorted, but there are, in turn, many examples. The following example is the same as the above, but this time the dictionary is a *SortedDictionary<Name, string>*:

```
static void Test11()
{
  IDictionary<Name, string> map = new SortedDictionary<Name, string>();
  map.Add(new Name("Knud", "Madsen"), "King");
  map.Add(new Name("Gudrun", "Jensen"), "Witch");
  map.Add(new Name("Svend", "Andersen"), "Warrior");
  map.Add(new Name("Olga", "Olesen"), "Soothsayer");
  Print(map);
}
```

and the difference is that the job positions are listed in another order determined by the persons last name. For it to works, the class *Name* must be changed to be *IComparable<Name>* so the keys can be sorted:

```
class Name : IComparable<Name>
{
  public string Firstname { get; set; }
  public string Lastname { get; set; }

  public Name(string firstname, string lastname)
  {
    Firstname = firstname;
    Lastname = lastname;
  }

  public int CompareTo(Name name)
  {
    if (Lastname.Equals(name.Lastname)) return
    Firstname.CompareTo(name.Firstname);
    return Lastname.CompareTo(name.Lastname);
  }

public override bool Equals(object obj)
  {
    if ((obj == null) || !this.GetType().
    Equals(obj.GetType())) return false;
    Name name = (Name)obj;
    return Firstname.Equals(name.Firstname) &&
    Lastname.Equals(name.Lastname);
  }
```

```
  public override int GetHashCode()
  {
    return Firstname.GetHashCode() ^ Lastname.GetHashCode();
  }

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

5.9 SORTEDLIST<K, T>

It is a collection which from the programmer's view is the same as a *SortedDictionary*, and internally is also a binary search tree. The difference is that a *SortedList* uses less memory than a *SortedDictionary*, and conversely, a SortedDictionary is more efficient in terms of inserting and deleting elements.

EXERCISE 7: AN INDEX

In this example, you must write a program that represents a keyword list for a book. The purpose is primarily to show how to use a *SortedDictionary*, but also a *LinkedList*.

A keyword consists of a name and number of page references, where a name is a string, while a page reference is an integer, and a keyword list (or index) is a list of keywords. The task is to write a program that can create an index and print the list on the screen.

Create a new console application project as you can call *IndexProgram*. Add a class *Index*. To the file with the class *Index* add another class which represent a keyword and a page references:

```
public class PageReferences
{
  private LinkedList<int> list = new LinkedList<int>();

  public string Name { get; private set; }
  public int Count
  {
    get { ... }
  }

  public PageReferences(string name, int page)
  {
    Name = name;
    list.AddFirst(page);
  }

  public int this[int n]
  {
    get { ... }
  }

  public int[] References
  {
    get { ... }
  }

  public void Add(int page)
  {
    ...
  }

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

where the first two properties returns the keyword and the number of page references. The method *Add()* must add a page reference, but such that the operation is ignored if the page number is already in the list, and if a page number is added the list must be ordered in increasing numbers.

ToString() must return a *string* that consists of the keyword and the page numbers separated by spaces.

To implement the indexer and the property *References()* you can do it trivial if you add the following using statement:

```
using System.Linq;
```

You should store the key words in a *SortedDictionary* with the name as key. It offers two benefits:

1. the list can be traversed and thus printed in order of the name
2. one can find page references to a particular keyword without searching, but by direct lookup

When comparing keys, you should not distinguishing between uppercase and lowercase letters. To solve that you can write your own key type:

```
class PageKey : IComparable<PageKey>
{
  public string Name { get; private set; }

  public PageKey(string name)
  {
  }

  public override bool Equals(object obj)
  {
  }

  public override int GetHashCode()
  {
  }

  public int CompareTo(PageKey key)
  {
  }

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

You can the write the class *Index*:

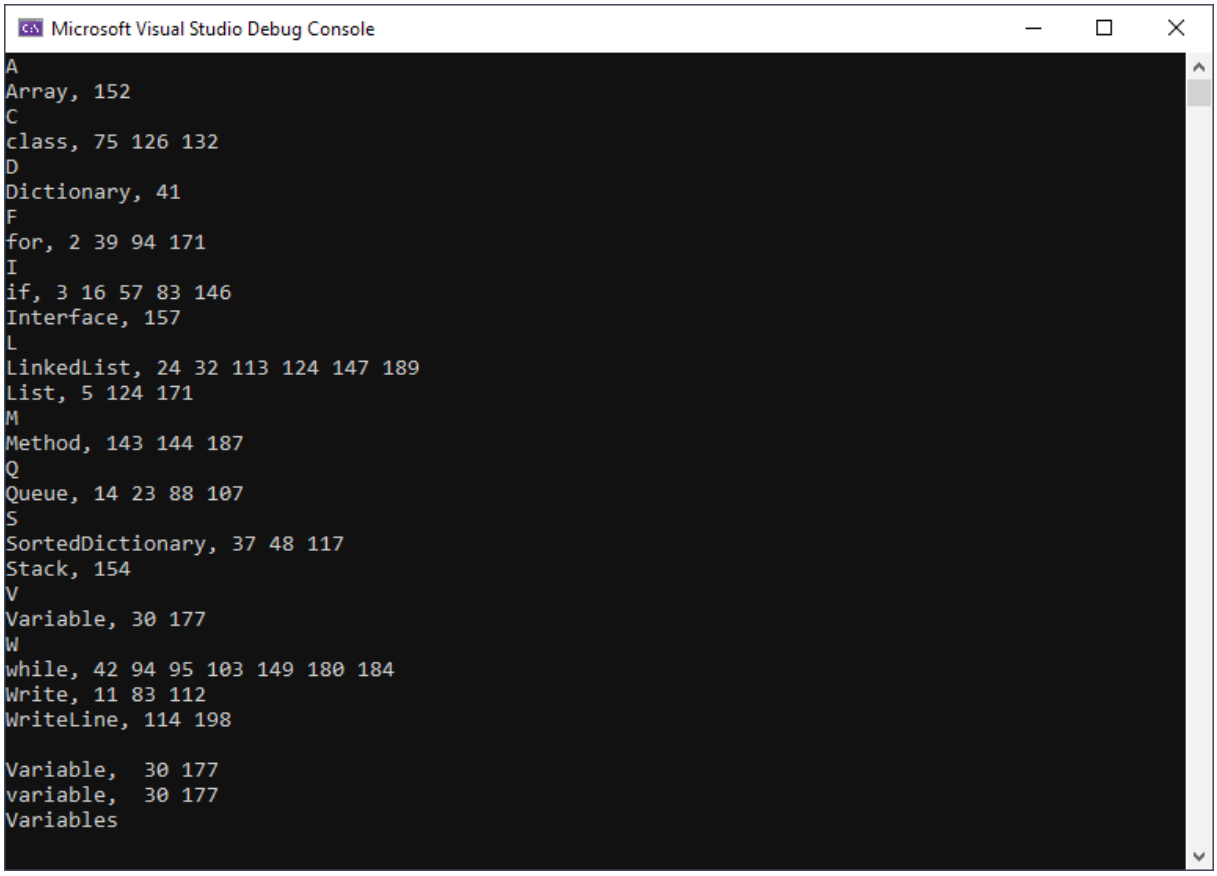
```
class Index
{
  private IDictionary<PageKey, PageReferences> table =
    new SortedDictionary<PageKey, PageReferences>();

  public bool Add(string name, int page)
  {
  }

  // Returns all page reference for a key work
  public int[] Pages(string name)
  {
  }

  // Prints the index
  public void Print()
  {
  }
}
```

You can test your class with the following program:

```
class Program
{
  static Random rand = new Random();
  static string[] words =
    { "Write", "if", "for", "while", "Variable",
      "WriteLine", "Array", "List",
     "Dictionary", "SortedDictionary", "Stack",
     "Queue", "LinkedList", "class",
     "Method", "Interface" };

  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();
    }
  }
```

and the result could be:



6 DELEGATES

A variable of a reference type can refer to an object of that type, which simply means that the variable contains the address of the location in the machine's memory where the object is located. Instead of reference, it is sometimes said that the variable points to the object and calls the variable a pointer, a word derived from the programming language C, but which actually gives a good explanation of what a reference is. When you have a reference to an object, you can manipulate the object by calling its methods and properties via the reference.

A delegate is also a reference type variable, but it is a variable that can contain a reference that can reference a method. This means that a delegate can point to a method and then you can execute the method using the reference. Immediately, it is not so easy to see what you can do with that, but it allows you to pass a method as a parameter to another method, and it can be used to write flexible code whose behavior can change dynamically at runtime.

You create a delegate with the following syntax:

```
public delegate int Calc(int a, int b);
```

Note that, apart from the word *delegate*, it looks like a signature for a method, but in fact it is a type. It is a type called *Calc*, and a variable of this type can refer to a method that has two *int* parameters and returns an *int*. The word *delegate* means that the compiler automatically and anonymously creates a class named *Calc*, which primarily defines three important methods:

```
sealed class Calc : System.MulticastDelegate
{
  public Calc (object target, uint functionAdress);
  public bool Invoke(int a, int b);
  public IAsyncResult BeginInvoke(int a, int
  b, AsyncCallback cb, object state);
  public bool EndInvoke(AsyncResult result);
}
```

These methods are not often referred to directly, but the class is a specialization of the class *System.MulticastDelegate*, which in turn is a specialization of *System.Delegate*. These classes contains a number of other methods and properties that are rarely used directly.

Below is shown two methods which both have the signature defined by the above delegate:

```
private static int Add(int a, int b)
{
  return a + b;
}

private static int Mul(int a, int b)
{
  return a * b;
}
```

The important thing is that both methods have the signature that a *Calc* variable can refer to, that is methods with two *int* parameters and which return an *int*. You can then create variables of the type *Calc* that refer to these methods:

```
Calc c1 = new Calc(Add);
Calc c2 = new Calc(Mul);
Console.WriteLine(c1(3, 7));
Console.WriteLine(c2(3, 7));
```

Here, a variable *c1* of type *Calc* is created and this variable refers to the method *Add()*. Similarly, *c2* is a delegate (of the same type) that refers to the method *Mul()*. The two methods can now be performed via the delegates, for example:

```
c1(3, 7)
```

In this case, the use of a delegate does not provide any benefits, as one could of course achieve the same simply by calling the two methods directly, and the example should then only serve to show the syntax of how to create and use a delegate.

There is no doubt that delegates are one of the slightly difficult C# concepts, and which at least, the first time you meet it, seems a bit mysterious. The term is closely related to events, and although it is a term that occurs in many contexts, events are especially something that comes with writing a program with a graphical user interface.

6.1 SIMPLE DELEGATES

I want to write a program that has a method with four parameters where the first three are integers, while the last is a delegate that reference a method for a calculation to be performed on the three first parameters. The method should print the three parameters and the result. It sounds a bit complicated to solve the task that way, but the goal is to write a method that has another method as a parameter, a problem that is precisely solved by a delegate.

I creates a console application project called *SimpleDelegate* where I defines a delegate with the following signature:

```
public delegate int Calc(int a, int b, int c);
```

It is a type called *Calc*, and a variable of this type can reference a method with tree *int* parameters and a method that must return an *int*. Now the desired method can be written as follows:

```
static void Calculation(int a, int b, int c, Calc Func)
{
  Console.WriteLine("Argument: " + a);
  Console.WriteLine("Argument: " + b);
  Console.WriteLine("Argument: " + c);
  Console.WriteLine("Calculation: " + Func(a, b, c));
}
```

As an example on a calculation method of the type *Calc()* the following method returns the sum of the three arguments:

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

And as another example the following method returns the maximum of the three arguments:

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

In *Main()* the method *Calculation()* can be performed as:

```
public static void Main(string[] args)
{
  Calculation(3, 7, 5, new Calc(Sum));
  Calculation (3, 7, 5, Max);
}
```

If you run the program the result is:

First, note how to define a delegate and including the reserved word *delegate*. In this case it is called *Calc*. Note that it is the name of a type, and you can create variables of this type. A delegate is a class type and can therefore also be used as a type for a parameter and a return type for a method. The method *Calculation()* is performed with *a*, *b* and *c* as parameters and a parameter for a method of the type delegate *Calc* called *Func*. Seen from the method *Calculation()* and the programmer who writes the method (and the compiler) you do not know which method *Func* refers to, but merely that it refers to a method that has three parameters of the type *int* and returns an int. The specific method is only determined at the time when *Calculation()* is called. In this case, the method *Calculation()* is a simple method, but in other contexts transferring a method as a parameter to another method can help to write a flexible and general code that can be used in many contexts. In this case, you can thus write the code for the method *Calculation()* without knowing exactly which calculation the method has to perform, in the same way that you do not know what values to calculate.

The method *Calculation()* is called from *Main()*:

```
Calculation(3, 7, 5, new Calc(Sum));
Calculation (3, 7, 5, Max);
```

Note the syntax for how to call the method. In the first case a delegate which refer to the method *Sum()* is created and is used as actual parameter to *Calculation()*. In the other case I just write the name of the method *Max()* and the three values it should works on. It is fine when the compiler use auto-boxing where the last parameter automatic is converted to a delegate, the compiler knows that the method *Calculation()* as parameter has a delegate of the type *Calc*, and when the method *Max()* have the correct signature, the compiler can perform the necessary conversion.

The above is really what there is to say about a simple delegate, for at least as long as talking only about syntax and semantics, and the most important thing here is *Calculation()*, which shows how to pass one method as a parameter to another method. In contrast, the above does not tell much about what a delegate can be used for and why it can be a good solution. This is the topic of the following, but note once that you can write the method *Calculation()* without knowing exactly how it should work, it is delegated to a method that is referenced via a parameter, which means, among other things, that how *Calculation()* works can change dynamically at runtime.

6.2 ENTER A NUMBER

As another example on a delegate I will write a program where the user must enter a number. The task is to write a method to enter the number when the method must

1. print a message on the screen that tells what for a number to enter
2. convert the entered text to an *int*, and then a number
3. test where the number is legal which depends on what the method is used for
4. repeat it all if step 2 or step 3 goes wrong

To solve step 3 the method must have a parameter for a method to perform the test, and that is a parameter which is a delegate:

```
public delegate bool Validate(int tal);
```

Thus, a type defined as a reference type, and a variable of this is a type is a function pointer that can point to a method that returns a *bool* and has an *int* as a parameter.

With this delegate available, the input method can be written as follows:

```
public static int Enter(string text, Validate Ok)
{
  text += " ";
  while (true)
  {
    Console.Write(text);
    try
    {
      int tal = Convert.ToInt32(Console.ReadLine());
      if (Ok(tal)) return tal;
    }
    catch
    {
    }
    Console.WriteLine("Illegal number, try again...");
  }
}
```

The method does not require much explanation, and the most important is how the parameter *Ok* is used to control the input. Also note that if the user enters something that is not an integer, then an exception is raised and the check method *Ok()* is not performed. You should also note that the input method does not know what the method *Ok()* does, but only that it returns true or false.

Below is shown a control method that is a method with the same signature as the delegate defines:

```
private static bool ThreeDigit(int t)
{
  return t > 99 && t < 1000;
}
```

An example to enter a three-digit integer could then be the following:

```
int t = Enter("Enter a 3-digit number ", ThreeDigit);
```

Similarly, below is shown a control method that tests whether an integer is a prime number:

```
private static bool IsPrime(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;
}
```

If the program needs to enter a prime number, it can be done as follows:

```
int p = Enter("Enter a prime number", IsPrime);
```

The result of all this is that by using a delegate you can write a more general input method, which only accepts numbers that are legal according to the validation function, and with a delegate, the code for the input function is separated from the control method.

6.2 MULTICAST DELEGATE

When you create a delegate (a delegate object), it will refer to a method. Internally, a delegate has a list that contains this reference, and a delegate can thus contain multiple references and refer to multiple methods. You are talking about a *multicast delegate* (note that a delegate inherits the class *System.MulticastDelegate*). Consider as an example the following delegate

```
public delegate void Print(string text);
```

that can refer to void methods with a string as a parameter. Below are three methods of this signature, where the first prints a *string* with the letters in the reverse order, the second prints a string with an extra space between all characters, and the third prints a string as the first character, the last character, then the second character and the next last character, and so on until all characters are printed:

```
private static void Reverse(string text)
{
  StringBuilder builder = new StringBuilder();
  for (int i = text.Length - 1; i >= 0; --i) builder.Append(text[i]);
  Console.WriteLine(builder);
}

private static void Split(string text)
{
  StringBuilder builder = new StringBuilder("" + text[0]);
  for (int i = 1; i < text.Length; ++i)
  {
    builder.Append(' ');
    builder.Append(text[i]);
  }
  Console.WriteLine(builder);
}

private static void Shift(string text)
{
  StringBuilder builder = new StringBuilder();
  for (int i = 0, j = text.Length; i < j; )
  {
    builder.Append(text[i++]);
    if (i < j) builder.Append(text[--j]);
  }
  Console.WriteLine(builder);
}
```

The following program creates a delegate that references the method *Reverse()*, and then it is also add references for the other two methods. The consequence is that when you execute the delegate, all three methods are executed. You must note the syntax for how to get a delegate to refer to multiple methods using the += operator (se chapter 8 about operator overriding). A delegate also has a method *GetInvocationList()*, which can be used to get the references to the methods referenced. Also note how to remove the reference to a method with the operator -=, and finally note that a delegate can refer to the same method several times (the method *Split()*). You should just accept the notation for add and remove references to a delegate, but it is explained below under operator override.

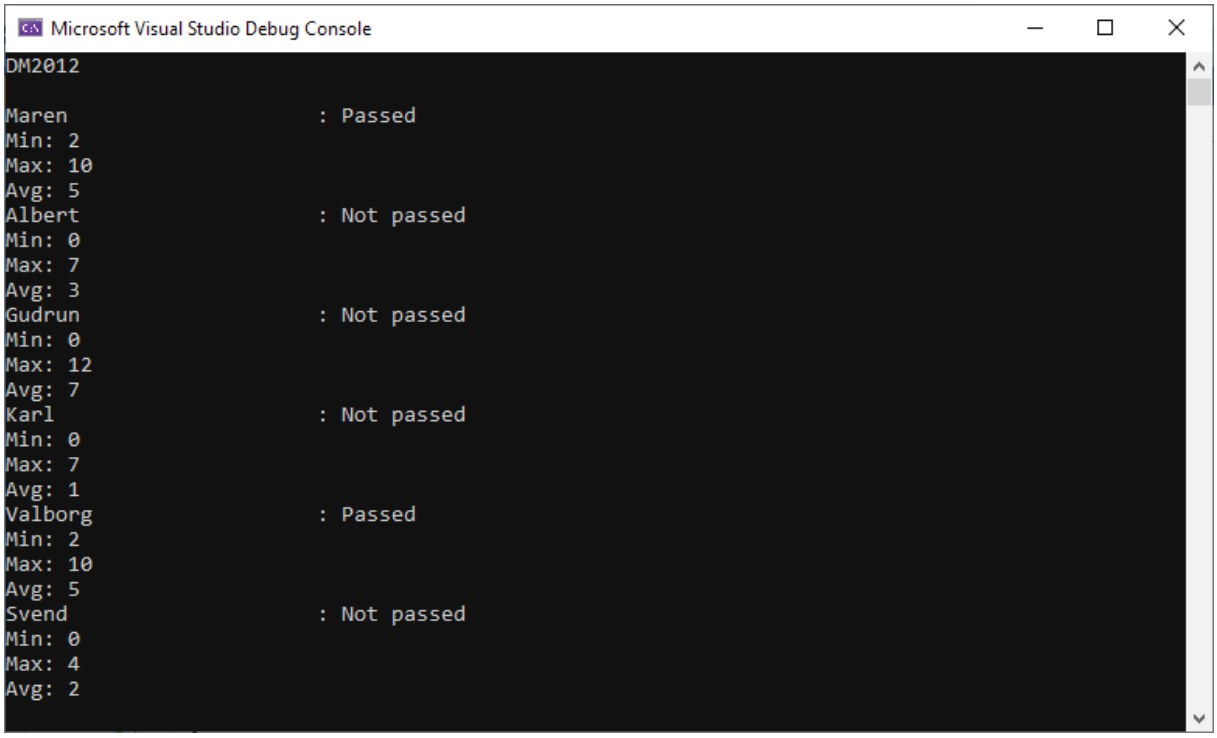
```
public static void Main()
{
  Print prn = new Print(Reverse);
  prn += Split;
  prn += Shift;
  prn("Hello World");
  foreach (Delegate d in prn.GetInvocationList())
  Console.WriteLine(d.Method.Name);
  prn -= Reverse;
  prn += Split;
  prn("Valdemar");
}
```

6.3 MORE ON DELEGATES

The theme for this example is students who have been given a number of grades. To make it simple, a student alone is defined by a name and then the grades that just are integers.

The program must create a class room with a number of students and the students must be awarded some grades. After that, the program must print a class list, where for each student some calculations are printed on the grades (such as the average), and whether the student has passed or not.

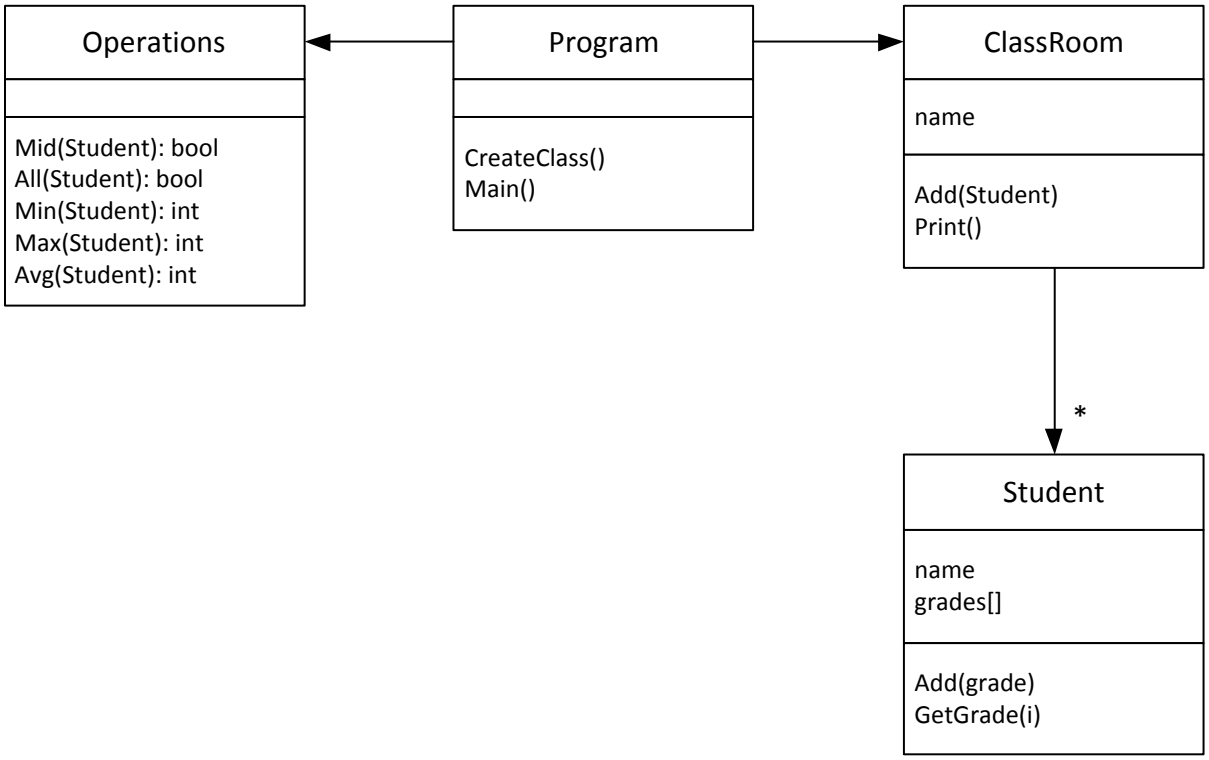
An example of a program run could be the following:



where a class room of 6 students has been printed. The print starts with the class name. For each student, a line is printed with the student's name and whether the student is passed or not. Next are three results:

- the smallest grade
- the greatest grade
- the average

The program is a simple console program. The program must work with students so there is a need for a class that can represent a student, and the class should essentially only allow one to add a grade as well as read the student's grades. There is also a need for a class which represents a classroom with students. Here, in addition to adding a new student, it should be possible to print a class list. It is the main program's task to create this classroom and add students, after which the program must print the class list by calling the method *Print ()* in the class *ClassRoom*. In response, the program can be outlined as follows:



The class *Operations* is explained below.

First, there is the class *Student*, which is a very simple class consisting solely of a string for the student's name and a list for the grades:

```
public class Student
{
  public string Name { get; private set; }
  private List<int> grades = new List<int>();

  public Student(string name)
  {
    Name = name;
  }

  public int Count
  {
    get { return grades.Count; }
  }
```

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

  public void Add(int grade)
  {
    grades.Add(grade);
  }

  public override string ToString()
  {
    StringBuilder builder = new StringBuilder(Name);
    foreach (int grade in grades) builder.Append(", " + grade);
    return builder.ToString();
  }
}
```

Next, there is the class *ClassRoom*, which also consists of a string for the name of the class, as well as a list of students. Here's the challenge the method *Print()*, which needs to know two things:

- what is the criterion for a student passing
- which calculations should be performed on a student's grades

Instead of writing methods for respectively checking whether a student has passed as well as specific calculation methods can you make the code more flexible by passing these methods as parameters to the method *Print()*. It requires two delegates:

```
public delegate bool Passed(Student stud);
public delegate int Calculation(Student stud);
```

where the first delegate represents a method that tests whether a student has passed, while the second represents a calculation method. That is the result of a calculation is an *int*. With these delegates available, the class *ClassRoom* can be written as follows:

```
public class ClassRoom
{
  public string Name { get; private set; }
  private List<Student> list = new List<Student>();

  public ClassRoom(string name)
  {
    Name = name;
  }

  public void Add(Student stud)
  {
    list.Add(stud);
  }

  public void Print(Calculation calculations, Passed Ok)
  {
    Console.WriteLine(Name + "\n");
    foreach (Student stud in list)
    {
      if (stud.Count > 0)
      {
        Console.WriteLine("{0, -25}: {1}", stud.Name,
          Ok(stud) ? "Passed" : "Not passed");
        foreach (Delegate opr in calculations.GetInvocationList())
          Console.WriteLine("{0}: {1}", opr.Method.
          Name, ((Calculation)opr)(stud));
      }
      else Console.WriteLine(stud.Name);
    }
  }
}
```

The class consists of the name for the classroom as well as a *List* for students, and there is a method that can add a student to the classroom (to the list). Then there is the *Print()* method, which has two delegates as parameters. The method starts by printing the class name, and then it performs a loop over all the students in the classroom. For each student, the name is printed and whether or not the student is passed. It is determined by the parameter *Ok*, which references a method for the passing criterion:

```
Ok(stud) ? "Passed" : "Not passed"
```

The method *Print()* and also the class *ClassRoom* do not know the passing criterion, and thus the method is decoupled and then the criterion. The *Print()* method has delegated the control for whether a student has passed to another object. This makes the method *Print()* more general and flexible.

Print() has another parameter *calculations*, which is also a delegate and which refers to calculation methods. *calculations* is used as a multicast delegate, running all the methods that the delegate refers to:

```
foreach (Delegate opr in calculations.GetInvocationList())
```

GetInvocationList() is a method that for a delegate return a list of the methods that the delegate refers to. The loop therefore runs through the methods that *calculations* points to. Note the type for *Calculation* that is an item in the list *GetInvocationList()* and then a *Delegate*.

Note that the relationship here is the same that *Print()* does not know which calculation methods to perform and it is determined when *Print()* is used.

The principle of delegating program logic to a method in another class is also called a *strategy pattern*.

Back there is the *Main* class, which has to create a class room with students and print the class list. With regard to the last thing, you have to send some calculation functions as well as a passing criterion with as parameters, and this is where the class *Operations* comes into the picture, as a class that defines two passing criteria and three calculation functions:

```
public static class Operations
{
  // Passing criterion, where a student is
  passed if the average is at least 2
  public static bool Mid(Student stud)
  {
    return Avg(stud) >= 2;
  }

  // Passing criterion, where a student is
  passed if all grades are at least 2
  public static bool All(Student stud)
  {
    for (int i = 0; i < stud.Count; ++i) if (stud[i] < 2) return false;
    return true;
  }

  // Calculation method which returns the average grade for a student
  public static int Avg(Student stud)
  {
    int sum = 0;
    for (int i = 0; i < stud.Count; ++i) sum += stud[i];
    return sum / stud.Count;
  }

  // Calculation method that returns the minimum grade for a student
  public static int Min(Student stud)
  {
    int min = stud[0];
    for (int i = 1; i < stud.Count; ++i) if
    (min > stud[i]) min = stud[i];
    return min;
  }

  // Calculation method, which returns the largest grade for a student
  public static int Max(Student stud)
  {
    int max = stud[0];
    for (int i = 1; i < stud.Count; ++i) if
    (max < stud[i]) max = stud[i];
    return max;
  }
}
```

The *Main()* method can then be written as:

```
public static void Main(string[] args)
{
  ClassRoom room = CreateClass();
  Calculation avg = Operations.Avg;
  Calculation min = Operations.Min;
  Calculation max = Operations.Max;
  room.Print(min + max + avg, Operations.All);
}
```

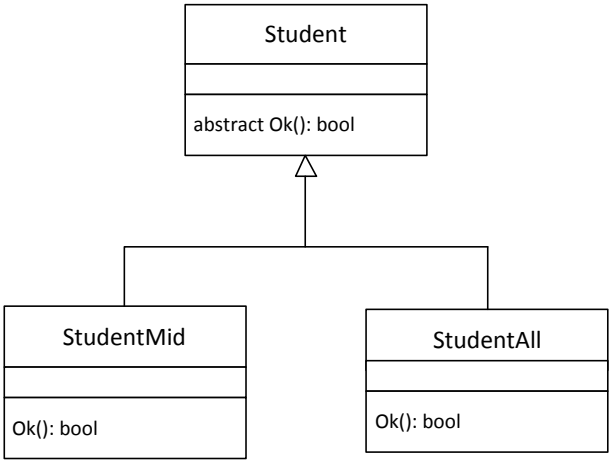
where *CreateClass()* is a method which creates a *ClassRoom* object and assigns 6 *Student* objects with associated grades. You should note how the method *Print()* is called:

```
room.Print(min + max + avg, Operations.All);
```

First, three delegates of the type *Calculation* are defined, which point to the three calculation methods in the class *Operations*. They are passed as parameters to the *Print()* method, and it is possible to write + between the variables, which means that the compiler will pass a multicast delegate referencing the three methods.

You must note that in the above examples, each time a delegate has pointed to a method where both the return type and parameters were simple types or a string. It is not necessary and both the return type and parameters can be arbitrary types.

The use of delegates is to some extent an alternative to inheritance. If you look at the delegation of the passing criterion above, you could instead have chosen a design as shown below, where there is a special kind of student for each passing criterion:



It is not always easy to determine which solution is the best, but generally you can say that delegates provide flexibility and the ability to write very general methods whose behavior can be modified at runtime, but also provides code that is more difficult to understand and correct.

If you consider a student with a passing criterion, then that criterion will rarely, if ever change. It is a static property for a particular kind of student, and it speaks to choosing an inheritance-based design. In practice, there will be few passing criteria and thus a need for a few specializations, and new passing criteria will rarely come, and other types of students will probably also differ on properties other than the exact passing criterion.

Relating to calculations are different. Here it can be difficult to know what calculations are needed and especially in the future, if new needs for calculations arise. Thus, it can be difficult to equip the *Student* class with the necessary calculation methods, and if new calculation methods often come up, it will be necessary every time to make a new specialization. If the calculation methods vary, it speaks to delegating rather than solving the problem through specialization.

6.4 GENERIC DELEGATES

A delegate can be generic, which simply means that it can refer to a generic method. As an example, I want to show a program, with a delegate that can reference a generic search method:

```
delegate int Search<T>(T[] arr, T elem);
```

By a search method, I want to understand a method that looks for an element in a generic array and returns the element's index. The delegate is called *Search*, but other than that, there is nothing that tells that it refer to a search method. The delegate refers to a method that, as a parameter, has an array of one type or another as well as an element of the same type, and a method that returns an int. In other words, it is a delegate who can refer to a generic method with this signature. As an example, two search methods with this signature are shown below:

```
static int LinSearch<T>(T[] arr, T elem)
{
  for (int i = 0; i < arr.Length; ++i) if
  (arr[i].Equals(elem)) return i;
  return -1;
}

static int BinSearch<T>(T[] arr, T elem) where T : IComparable<T>
{
  for (int a = 0, b = arr.Length - 1; a <= b; )
  {
    int m = (a + b) / 2;
    if (arr[m].Equals(elem)) return m;
    if (elem.CompareTo(arr[m]) < 0) b = m - 1; else a = m + 1;
  }
  return -1;
}
```

In this context, the algorithms are not important, but the first implements a regular linear search and works by traversing the array from start to finish. If the element is found, the method returns the index, and if you reach through the entire array, you can conclude that the element does not exist and the method returns -1 instead. The second method has the same signature and performs the same, but instead implements binary search. The requirement here is that the array is already arranged (sorted) and must therefore be parameterized with a type that is comparable. The principle of binary search is that you start by comparing with the middle element, and if it is the element that is searched, you return the index. Otherwise, you take advantage of the array being arranged, so you know whether to search the left or right half. Proceed as follows so that for each repetition you either find the element or halve the number of elements to search.

You should note that although binary search is parameterized with a comparable type, it still has a signature so that it can be referenced by a *Search* delegate.

The following is a *Main()* method which uses the *Search* delegate and the above search methods:

```
static void Main(string[] args)
{
  int[] v = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };
  Find(v, 31, LinSearch);
  Find(v, 32, LinSearch);
  Find(v, 31, BinSearch);
  Find(v, 32, BinSearch);
}

static void Find(int[] arr, int t, Search<int> f)
{
  int n = f(arr, t);
  if (n < 0) Console.WriteLine("{0} is not found", t);
  else Console.WriteLine("{0} is found at index {1}", t, n);
}
```

EXERCISE 8: ENTER AN OBJECT

In this exercise you must write, in the the same way as in the section *Enter number*, a program with an enter method, but this time a method for enter an object of any type.

Create a new console application project that you can call *EnterObject* and add the following delegate to the project:

```
public delegate bool Control<T>(T arg);
```

It defines a method used to validate the entered object and works in the same way as in the previous example, but this time it is a generic delegate.

You should then define the input method as:

```
public static T Enter<T>(string text,
Converter<string, T> Convert, Control<T> Ok)
{
}
```

It is a generic method with three parameters, where the first is the text written on the screen and the last is as in the previous example a method used to validate the entered value. The second parameter also refer to a method, but a method used to convert the entered value (which is always a *string*) to an object of the type *T*. The framework has a lot of predefined delegates and *Converter* is one of them:

```
public delegate TOutput Converter<TInput, TOutput>(TInput input);
```

In this case *TInput* is *string* and *TOutout* is the generic type *T*.

When you have written the method you must test it using it to enter a 3-digit integer and a prime number that is like in the previous example. Note that you can use *Convert.ToInt32()* as conversion method.

When it seam to work you must add the following type to your project:

```
public struct Point
{
  public double x;
  public double y;

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

  public override string ToString()
  {
    return string.Format("({0:F4}, {1:F4})", x, y);
  }
}
```

You must then enter two *Point* objects (for example as the two coordinates separated by space), where the fist point must be in first quadrant (both coordinates must be positive) and the second point must be within the unit circle (its distance to (0, 0) must be less than or equal to 1). Note that this means that you must write two validation methods and a conversion method which can convert a *string* to a *Point*.

6.5 ASYNCHRONOUS DELEGATE

In this section I will mention asynchronous delegates, but this is something I will return to in a later book (C# 8) as it is based on threads. A thread is some code (a method) that can be performed in parallel with the rest of the program and in that way users feel that the program is doing several things at the same time.

When the method to which a delegate refers is performed, it is the same as with a normal function call: The control is transferred to the method and the method retains control until it terminates, after which the program continues from where the call occurred. It is said that the method is executed synchronously with the program. With a delegate, it is possible to start a method, while the main program itself continues its work in parallel with the method being executed. This corresponds to the method referred to by the delegate runs in its own thread.

The following program is called *Sorting* and is a program used to sort an array, but the sorting must be done in parallel with the main program, which continues with its work. The sorting must be done asynchronously. You must note that the program (the project) is created as a

Console App (.NET Framework)

It is important to include references to the needed assemblies. To sort the array the following method is used:

```
public static void Sort(int[] t)
{
  Console.WriteLine("Start sort: " + DateTime.Now.ToLongTimeString());
  for (int i = 0; i < t.Length - 1; ++i)
  {
    int k = i;
    for (int j = i + 1; j < t.Length; ++j)
      if (t[k] > t[j]) k = j;
    if (k != i)
    {
      int u = t[i];
      t[i] = t[k];
      t[k] = u;
    }
  }
  Console.WriteLine("Sort terminated: " +
  DateTime.Now.ToLongTimeString());
}
```

The important thing is not the sorting algorithm, but merely that it is a method that takes time for large arrays. Below is also shown a delegate that can refer to methods with the same signature as *Sort()*:

```
public delegate void Function(int[] t);
```

The next method calls the sort method:

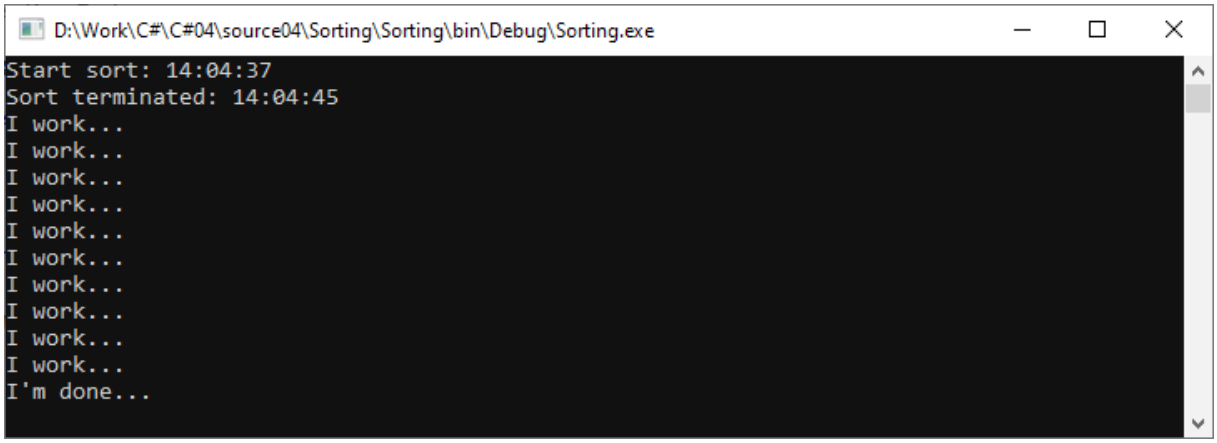
```
public static void Test1()
{
  int[] t = Create(80000);
  Function f = new Function(Sort);
  f(t);
  for (int i = 0; i < 10; ++i)
  {
    Console.WriteLine("I work...");
    System.Threading.Thread.Sleep(1000);
  }
  Console.WriteLine("I'm done...");
}
```

Here is *Create()* a method which creates an array with the actual number of elements. Also note the statement:

```
System.Threading.Thread.Sleep(1000);
```

which means that the method *Test1()* is suspended for 1000 milliseconds and thus for 1 second. That is *Test1()* waits 1 second before continuing.

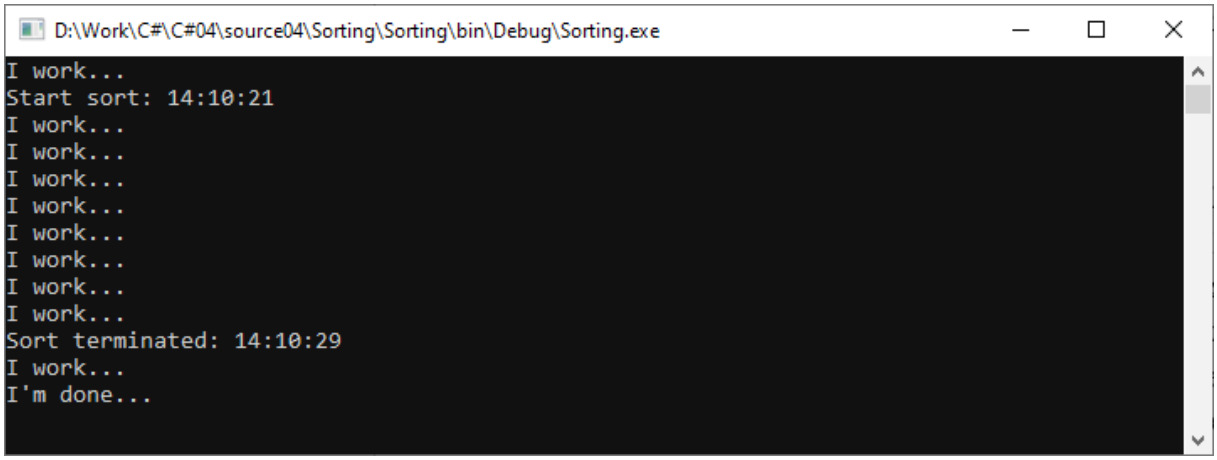
If the method is carried out, the result may be:



The result is not very surprising: First, *Sort()* is executed, and then *Test1()* continues with its work. This means that *Sort()* and *Test1()* are performed synchronously.

However, it is possible to execute the method *Sort()* with an asynchronous delegate, which simply means that the method is executed in its own thread parallel to the *Main()* method. This is done in the following method, which is basically the same method as *Test1()*:

```
public static void Test2()
{
  int[] t = Create(80000);
  Function f = new Function(Sort);
  IAsyncResult r = f.BeginInvoke(t, null, null);
  for (int i = 0; i < 10; ++i)
  {
    Console.WriteLine("I work...");
    System.Threading.Thread.Sleep(1000);
  }
  f.EndInvoke(r);
  Console.WriteLine("I'm done...");
}
```



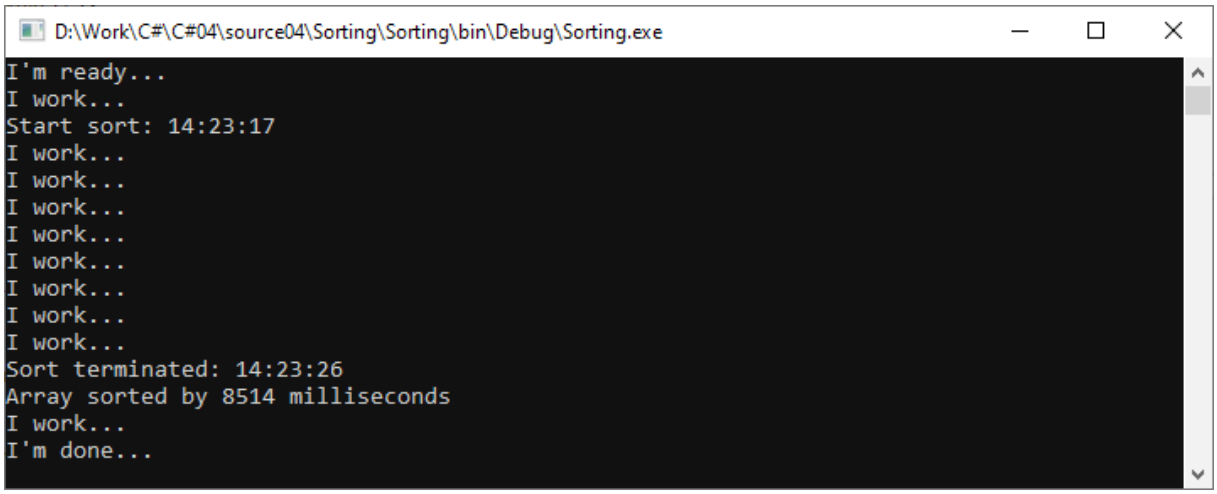
The difference is that the method referred to by the delegate, here the method *Sort()*, is started with *BeginInvoke()*, which in short means that the method is started in its own thread and thus runs asynchronously parallel to the main thread. The first parameter for *BeginInvoke()* is the parameter for the method *Sort()* method and the others are explained below. Note that if the delegate had referred to a method with more parameters, then a corresponding number of current parameters should be specified at that location. Finally, *EndInvoke()* is executed with the return value from *BeginInvoke()* as a parameter. This means that the main thread waits until the *Sort()* method is complete. This is necessary, because if the main thread terminates before the *Sort()* method is complete, its thread will also end, and method *Sort()* will not run to the end. In this case, *EndInvoke()* will be void because the delegate is void, but if the delegate references a method with a return value, *EndInvoke()* will return this value.

BeginInvoke() has two additional parameters. The first is a callback method, actually a delegate, that is a method that is executed when the method started by the delegate terminates. The last is an argument of the type *object* that can be passed as a parameter to the callback method. The type of callback method is *AsyncCallback*, which is a delegate to a void method with an argument of type *IAsyncResult*. An example is

```
public static void Sorted(IAsyncResult res)
{
  DateTime start = (DateTime)res.AsyncState;
  TimeSpan time = DateTime.Now.Subtract(start);
  Console.WriteLine("Array sorted by {0} milliseconds",
    (int)time.TotalMilliseconds);
}
```

Consider the following version of the test method:

```
public static void Test3()
{
  int[] t = Create(80000);
  Console.WriteLine("I'm ready...");
  Function f = new Function(Sort);
  IAsyncResult r = f.BeginInvoke(t, new
  AsyncCallback(Sorted), DateTime.Now);
  for (int i = 0; i < 10; ++i)
  {
    Console.WriteLine("I work...");
    System.Threading.Thread.Sleep(1000);
  }
  Console.WriteLine("I'm done...");
  f.EndInvoke(r);
}
```



Basically, the method is identical to *Test2()*, but this time you specify a callback method that is executed when the method that the delegate refers to (the sort method) is terminated and a parameter is also sent to the callback method.

6.6 CALLBACK

A (running) program consists of a number of objects that work together to solve the desired task. An object is usually created in a method (for example, in *Main()*), and then you can send messages to the object by calling its methods. For example creates Main() a *B* object and sends a message to it:

```
class B
{
  void Message(string txt)
  {
    ....
  }
}

class A
{
  public static void Main()
  {
    B obj = new B();
    obj.Message("hi");
  }
}
```

It is that the *A* object can communicate with the *B* object. However, communication is often needed in both directions, that is where the *B* object wants to send a message back to the *A* object e.g. to tell the object that some event has occurred. It is usually called callback and can be implemented in several ways.

The project *Callback1* implements callback using a reference to the object to which the callback is to be made, and that means where the calling object transfers a reference to itself to the called object so that the called object can call back.

The task is to write a program that, in addition to the *Main()* class, has two classes, one representing a watch, while the other representing a display showing the value of the watch when:

- the watch creates the display and the knows the name of the display
- the display has to read the watch and then must know the watch

The two classes are:

```
class Program
{
  static void Main(string[] args)
  {
    new Watch();
    for (; ; )
    {
      Console.WriteLine("Main....");
      System.Threading.Thread.Sleep(3000);
    }
  }
}

class Watch
{
  private Display display;

  public Watch()
  {
    display = new Display(this);
    (new Action(Go)).BeginInvoke(null, null);
  }

  public void Go()
  {
    for (; ; )
    {
      System.Threading.Thread.Sleep(1000);
      display.Refresh();
    }
  }

  public string Time
  {
    get { return DateTime.Now.ToLongTimeString(); }
  }
}
```

```
class Display
{
  private Watch watch;

  public Display(Watch watch)
  {
    this.watch = watch;
  }

  public void Refresh()
  {
    Console.WriteLine("The time is " + watch.Time);
  }
}
```

The *Main()* method runs an infinite loop, where it prints a message every 3 seconds on the screen. It is just for the program to simulate that something is happening while the clock is running.

The class *Watch* should simulate a simple watch. The watch should use a display represented by the class *Display*. The watch creates a display and thus knows the *Display* object. The method *Go()* runs the watch and is started in the constructor in the class *Watch*. It starts as an asynchronous delegate and runs in an infinite loop that updates the display every second, and note that it can do that as the watch knows the display through the reference *display*.

Also note how to start the watch with the delegate *Action*. It is a generic delegate defined by the framework, which can refer to a void method without parameters. *Action* is also found in several overrides. Also note that no *EndInvoke()* is performed this time. The reason is that *Main()* runs an infinite loop and therefore does not terminate the thread running the method *Go()*.

The method *Go()* calls the method *Refresh()* on the object *Display*, which should print what the watch shows. The object *display* must therefore be able to read the watch and thus know the *Watch* object, that is the calling object. Therefore, the object *Display* gets a reference to the *Watch* object in the constructor so that it knows the *Watch* object.

The task could be solved in other and simpler ways (for example could the method *Refresh()* in the class *Dipslay* have a parameter for the time), but the goal here is to give an example of two classes that communicate by knowing each other.

There is nothing wrong with the above form of two-way communication and it is a way that is often used in practice. However, the solution has the problem of coupling the two classes very strongly, since the class *Watch* must know the class *Display*. If another class of another type (another display) also needs a *Watch* object, it will cause a problem as it cannot immediately be notified that the watch is updated.

The project *Callback2* is doing exactly same as *Callback1*, but this time the problem is solved, so that the individual *Display* objects are notified that the watch has ticked, but without the watch knowing the *Display* objects. The code is changed as follows:

```
namespace Callback2
{
  public delegate void TimeObserver();

  class Program
  {
    static void Main(string[] args)
    {
      Watch watch = new Watch();
      new Display(watch);
      for (; ; )
      {
        Console.WriteLine("Main....");
        System.Threading.Thread.Sleep(3000);
      }
    }
  }

  class Watch
  {
    private TimeObserver observers;

    public Watch()
    {
      (new Action(Go)).BeginInvoke(null, null);
    }

    public void Go()
    {
      for (; ; )
      {
        System.Threading.Thread.Sleep(1000);
        if (observers != null) observers();
      }
    }
```

```
    public void AddObserver(TimeObserver obs)
    {
      observers += obs;
    }

    public string Time
    {
      get { return DateTime.Now.ToLongTimeString(); }
    }
  }

  class Display
  {
    private Watch watch;

    public Display(Watch watch)
    {
      this.watch = watch;
      watch.AddObserver(Refresh);
    }

    public void Refresh()
    {
      Console.WriteLine("Klokken er nu " + watch.Time);
    }
  }
}
```

First, note that a delegate named *TimeObserver* is defined. The class *Watch* has a variable of this type and a method *AddObserver()* that can add a method to the *TimeObserver* object. Then there is the method *Go()* that runs the watch the same way as before, and so it ticks every second. Each time the watch is ticking, it calls the methods that the delegate *observers* refer to, but note that the method *Go()* does not know if there are any objects on the other end or what it is for objects, whether they are *Display* objects or something other things. The method *Go()*alone knows that if there are any "listeners", that is someone that has a method with the signature that the delegate *TimeObserver* defines, and the method *Go()* does not decide what these methods do. The result is that the class *Watch* and the class *Display* are very loosely coupled.

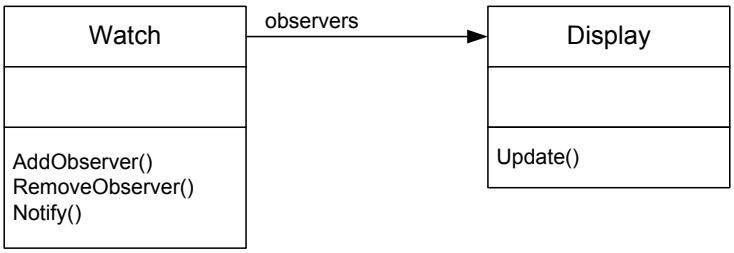
The class *Display* is almost unchanged, but the constructor signs up for the watch as an observer. That is that the constructor calls the method *AddObserver()* in the class *Watch*.

Note that a *Watch* can very well have multiple observers, since *TimeObserver* is a multicast delegate.

6.7 THE OBSERVER PATTERN

The following example concerns same problem as above, and illustrates a programming principle called an observer server pattern. In relation to the class *Watch* and the class *Display*, the problem can be described as follows:

In an object of a class often called *subject* (the *Watch*), an event may occur (here the watch is ticking). When this happens, the object must send notifications to *observer* objects (*Display* objects) that the event has occurred, the *subject* object must call a method on the *observer* objects (the method *Refresh()*) so that they can do what is necessary. To that to be possible, the object uses a delegate to keep track of its observers, and the only thing the subject object knows is that the observer objects have a method with an agreed signature that is defined in the form of a delegate. In addition, it must be possible for observer objects to subscribe or unsubscribe as being observer objects, and correspondingly the pattern can be illustrated as:



The delegate *TimeObserver* is as above, but there is some changes in the class *Display*:

```
class Display
{
  private Watch watch;
  private string name;
  private bool started = false;
  private TimeObserver refresh;

  public Display(string name, Watch watch)
  {
    this.name = name;
    this.watch = watch;
    refresh = new TimeObserver(Refresh);
  }
```

```
  public void Start()
  {
    if (!started)
    {
      watch.AddObserver(refresh);
      started = true;
      Console.WriteLine(name + " started");
    }
  }

  public void Stop()
  {
    if (started)
    {
      watch.RemoveObserver(refresh);
      started = false;
      Console.WriteLine(name + " stopped");
    }
  }

  public void Refresh()
  {
    Console.WriteLine("{0} siger at klokken er {1}", name, watch.Time);
  }
}
```

The class has been given three additional variables. The variable *name* is only for the *Refresh()* method to print a name so that you can see which *Display* object receives a notification. The variable *started* is used to control whether a *Display* is subscribed to the *Watch* object as an observer and thus should receive notifications, and it is used solely to control that the same *Display* object does not register as an observer multiple times. The last variable is a delegate that refers to the *Refresh()* method and is a reference sent to the *Watch* object for a *Display* object to subscribe as an observer or unsubscribe.

In addition, the class has a special method *Start()* that a *Display* object can use to register as an observer. Here you should note that the method changes the value of the variable *started* and that it writes a message on the screen that it is now registered as an observer. It only happens if the object is not already an observer.

The method *Stop()* is completely parallel to the method *Start()* except that it instead unsubscribe an object as an observer.

Then there is the class *Watch* that is virtually unchanged, except that it now has a *RemoveObserver()* method:

```
class Watch
{
  private TimeObserver observers;

  public Watch()
  {
    (new Action(Go)).BeginInvoke(null, null);
  }

  public void Go()
  {
    for (; ; )
    {
      Thread.Sleep(1000);
      if (observers != null) observers();
    }
  }

  public void AddObserver(TimeObserver obs)
  {
    observers += obs;
  }

  public void RemoveObserver(TimeObserver obs)
  {
    observers -= obs;
  }

  public string Time
  {
    get { return DateTime.Now.ToLongTimeString(); }
  }
}
```

Below is a test program

```
static void Main(string[] args)
{
  Watch watch = new Watch();
  Display[] displays =
    {new Display("Svend", watch),
     new Display("Knud", watch),
     new Display("Valdemar", watch) };
  while (true)
  {
    Thread.Sleep(rand.Next(3000, 5000));
    bool start = rand.Next(2) == 1;
    Display d = displays[rand.Next(displays.Length)];
    if (start) d.Start(); else d.Stop();
  }
}
```

First, a *Watch* object is created. Next, an array of three *Display* objects are created which are observer objects, but since their *Start()* method is not called, the objects are not yet receiving notifications. The program now runs in an infinite loop, waiting for each repeat randomly between 3 and 5 seconds. Next, it is chosen at random whether the program should register or unsubscribe an observer for the watch, and the corresponding operation is now performed at a random of the three *Display* objects. The result is that the program simulates a situation where you randomly subscribe and unsubscribe observes for the watch.

EXERCISE 9: TEMPERATURE

You must write a program which is similar to the above regarding the observer pattern, but this time the subject should be a thermometer, and there could be one or more observes for the thermometer.

Start with a new *Console App (.NET Framework)* project as you can call *AlarmProgram*.

Add a class *Thermometer* to represent the thermometer, but this time there must be three kind of observers for

- if the temperature is changed (the thermometer measure the temperature)
- if the temperature exceeds 80 degrees where a warning must be reported
- if the temperature reaches 100 degrees where an alarm should be sent and the thermometer stops

Defines delegates for these observers, when the delegate for the temperature change must have a parameter showing the temperature.

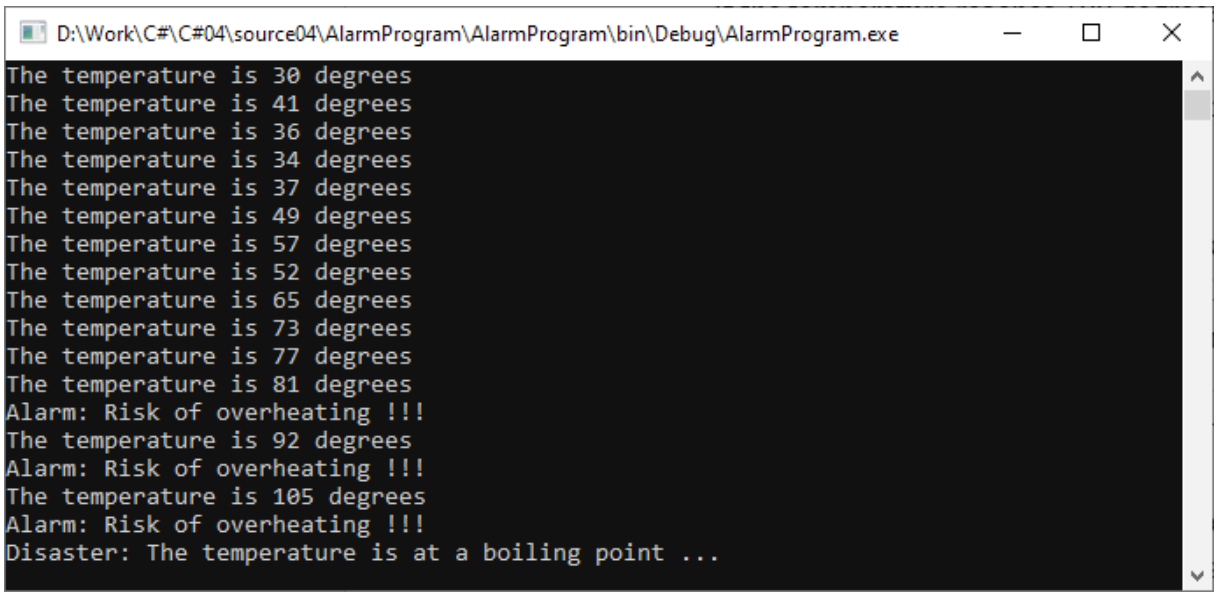
The class *Thermometer* must starts (in the constructor) by setting the temperature to a random value between 10 and 20 degrees, and it must keep track of the temperature with a variable *temperature*. It then starts a method *Go()* asynchronous that simulates that the temperature is measured every second and every second the temperature changes with a random value between -5 and 20 degrees. Each time the temperature is measured observes must be notified.

You must then add a class *Display* that is observer for the thermometer, when a *Display* object must be observer for all three kinds of notifications. The constructor must have a *Thermometer* object as parameter, so it can register as observer.

The *Main()* method could be:

```
static void Main(string[] args)
{
  Termometer termometer = new Termometer();
  new Display(termometer);
  while (true) Thread.Sleep(100);
  Console.ReadLine();
}
```

and an example of running the program could be:



6.8 EVENTS

If you look at how to use a delegate for callback, then the class that needs to execute the callback - the subject class - must have methods where listeners (observer objects) can register and unsubscribe. All that can be automated by an event that really just means that this registration logic is auto generated. I want to illustrate events with the above callback example, but events are very closely linked to programs with a graphical user interface.

The delegate *TimeObserver* is as above, but the class *Watch* has changed so that instead of a variable of the type *TimeObserver* it has an event of this type:

```
class Watch
{
  public event TimeObserver Observers;

  public Watch()
  {
    (new Action(Go)).BeginInvoke(null, null);
  }

  public void Go()
  {
    for (; ; )
    {
      Thread.Sleep(1000);
      if (Observers != null) Observers();
    }
  }

  public string Time
  {
    get { return DateTime.Now.ToLongTimeString(); }
  }
}
```

First, note that it is a *public* variable. Then note that the methods for adding and removing an observer are gone. They are unnecessary since the variable *observers* are *public*.

The class *Display* is almost unchanged, except that it is now the two methods *Start()* and *Stop()* that are directly respectively used to subscribes and unsubscribes:

```
public void Start()
{
  if (!started)
  {
    watch.Observers += refresh;
    started = true;
    Console.WriteLine(name + " started");
  }
}

public void Stop()
{
  if (started)
  {
    watch.Observers -= refresh;
    started = false;
    Console.WriteLine(name + " stopped");
  }
}
```

After that, the program can run and perform exactly the same as before.

Immediately, an event does not look like much more than a public delegate, and that is actually what it is, but there is some form of standard that one should adhere to. An event is associated with an underlying *void* delegate with two parameters, the first which has the type *object* and is a reference to the object that sends the notification, while the second has the type *System.EventArgs* and is used to pass arguments to the recipient. *System.EventArgs* has no fields and is only used as the base class for specific event arguments. In most cases, it is recommended to follow this signature for events as this ensures consistent event handling.

To show event args in action I will look at a last version of the callback program called *Callback5* where I have added a class

```
public class TimeEventArg : EventArgs
{
  private DateTime time;

  public TimeEventArg(DateTime time)
  {
    this.time = time;
  }

  public DateTime Time
  {
    get { return time; }
  }
}
```

There is not much to explain, but note that the type inherits *EventArgs*, which is what tells the type to be used as a parameter for an event handler. In this case, an object will represent a time, which is the value passed to the event handler.

The Delegate *TimeObserver* must then be changed:

```
public delegate void TimeObserver(object sender, TimeEventArg e);
```

You have to think of it as a type that defines an event handler - a method that must be executed when an event occurs, and an event in this case will mean that the clock is ticking.

Then the class *Watch* must be modified:

```
class Watch
{
  public event TimeObserver Observers;

  public Watch()
  {
    (new Action(Go)).BeginInvoke(null, null);
  }
```

```
  public void Go()
  {
    for (; ; )
    {
      Thread.Sleep(1000);
      if (Observers != null) Observers(this,
      new TimeEventArg(DateTime.Now));
    }
  }
}
```

It now no longer has any property that observers can use to read the watch, but when the watch is ticking, it sends notifications to observers and along with these notifications the time is sent in the form of a *TimeEventArg* object. The result is that the objects event handlers receive the time as a parameter.

In the class *Display* there is only one change, the method *Refresh()* (that is, the event handler), which now has two parameters:

```
public void Refresh(object sender, TimeEventArg e)
{
  Console.WriteLine("{0} siger at klokken er {1}", name, e.Time);
}
```

Note that the first parameter is not used for anything, but it is recommended to include it as it is standard for event handlers, and there are examples of uses.

Instead of define the delegate *TimeObserver* for the event handler .NET has a generic type *EventHandler* to define event handlers and you could instead have written:

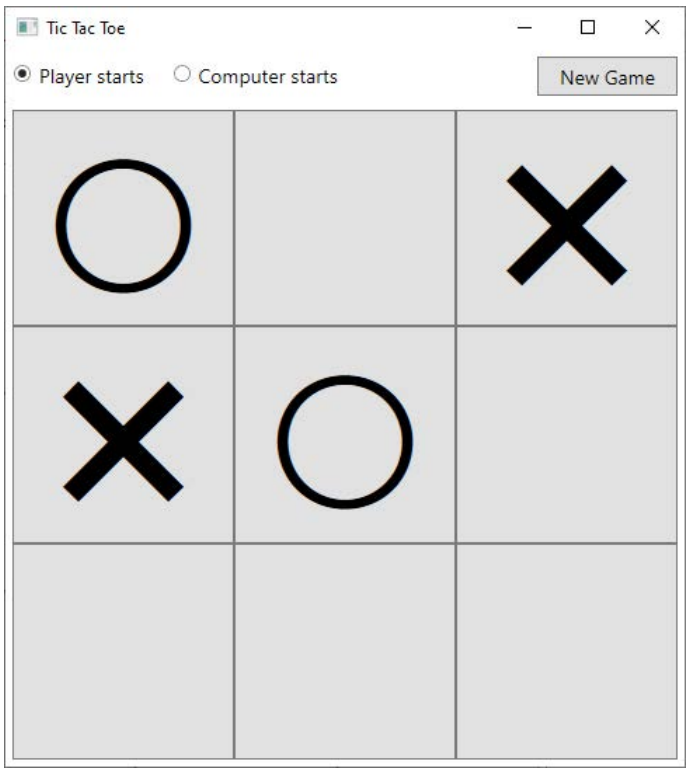
```
public event EventHandler<TimeEventArg> Observers;
```

and it is generally recommended to use this type for event handlers.

PROBLEM 4: TIC TAC TOE

You must write a program that simulates the classic Tic Tac Toe game. The user interface could be:

where there are two checkboxes to select where the user or the computer should start, and there is also a button to select a new game. The rest of the window is filled with 9 buttons, and below is a window, where the user has clicked two buttons (the cross) and the computer has selected the other two (the circles).



You should write the program as a class *MainWindow* for the user interface and another class which you can call *Game*. This class must implement the program logic and check if the user clicks on a legal button, but it is also the class which has to determine the computer's choice. When the class *Game* sees that there is three symbols on a line (horizontal, vertical or diagonal) the class should fire an event, and the main window should subscribe as listener for this event. The event must have an argument which tells whether it is the user or the computer having three in a row or whether it is because all cells are filled.

The biggest challenge in writing the program is choosing a good algorithm for how the computer should select a cell. Here you can let your computer choose a random empty cell, but that means there is a good chance that you can always win over the computer. A better solution is to try and find a better algorithm on the internet that you can apply, and there are an incredible number of solutions, so there is something to choose from. Incidentally, in one of the last books in this series, I will return to the algorithm.

7 LAMBDA EXPRESSIONS

Lambda expressions are a short notation for an anonymous method. Consider the following methods:

```
public static void Test01()
{
  List<int> list = new List<int>();
  list.AddRange(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
  Predicate<int> callback = new Predicate<int>(IsEven);
  List<int> even = list.FindAll(callback);
  foreach (int t in even) Console.WriteLine(t);
}

public static bool IsEven(int n)
{
  return n % 2 == 0;
}
```

First, a list of 9 numbers are defined. The type *List* has a method *FindAll()*, which returns a partial list of the elements that meet a specific criterion. The criterion is specified as a parameter to *FindAll()* in the form of a delegate, which returns a *bool* and has an argument of the list's parameter type. The method *IsEven()* is thus a criterion. Correspondingly, a delegate callback is defined, which points to the method *IsEven()*, and is then used as a parameter to *FindAll()*. The result is that the method *Test1()* prints all even numbers in the list.

Since I have defined anonymous methods above, the problem can also be solved as follows:

```
public static void Test02()
{
  List<int> list = new List<int>();
  list.AddRange(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
  List<int> even = list.FindAll(delegate(int n) { return n % 2 == 0; });
  foreach (int t in even) Console.WriteLine(t);
}
```

where the parameter to *FindAll()* this time is an anonymous delegate. In cases where the criteria method is as simple as above, writing an anonymous method is simpler than defining a specific delegate that can reference a method.

Finally, you can solve the problem with a *lambda* expression:

```
public static void Test03()
{
  List<int> list = new List<int>();
  list.AddRange(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
  List<int> even = list.FindAll(n => n % 2 == 0);
  foreach (int t in even) Console.WriteLine(t);
}
```

This time, the the parameter to *FindAll()* is an expression, a lambda expression. In fact, based on the parameter type *Predicate<T>*, which is a delegate, the compiler will convert the expression to an anonymous method.

Basically, a lambda expression consists of a parameter list followed by a lambda operator => and again followed by one or more statements:

- a parameter list, here *n*
- a lambda operator =>
- statements, here `n % 2 == 0`

You can explicitly specify the type of the parameters, for example

```
(int n) => n % 2 == 0
```

but in this case it is not necessary as the compiler can see from the list that the type is *int*. Note that if you specify an explicit parameter type, then the type and name must be in parentheses.

The statement in a lambda expression need not only be a simple statement, but it must also be a block. For example the following method will print all prime numbers in a list:

```
public static void Test4()
{
  List<int> list = new List<int>();
  list.AddRange(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
  List<int> prims = list.FindAll(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;
  });
  foreach (int t in prims) Console.WriteLine(t);
}
```

It may also be an example of where one should not use a lambda expression, as the above is difficult to read, but in other cases where there are only two or three simple statements, a block is an excellent solution.

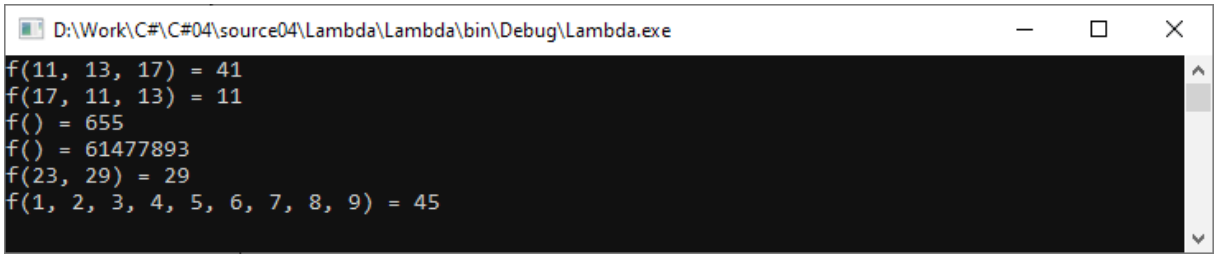
There are several variants of lambda expressions. Consider the following methods:

```
public static void Test5()
{
  Print((a, b, c) => a + b + c, 11, 13, 17);
  Print((a, b, c) => { int t = a < b ? a : b;
  return t < c ? t : c; }, 17, 11, 13);
  Print(() => rand.Next(1000));
  Print(() =>
  {
    DateTime time = DateTime.Now;
    return time.Hour * 3600000 + time.Minute
    * 60000 + time.Second * 1000 +
          time.Millisecond;
  });
  Print((int[] a) => (a[0] < a[1] ? a[1] : a[0]), 23, 29);
  Print((int[] a) => { int s = 0; foreach
  (int t in a) s += t; return s; },
        1, 2, 3, 4, 5, 6, 7, 8, 9);
}
```

```
public static void Print(Function3 f, int a, int b, int c)
{
  Console.WriteLine("f({0}, {1}, {2}) = {3}", a, b, c, f(a, b, c));
}
public static void Print(Function0 f)
{
  Console.WriteLine("f() = {0}", f());
}

public static void Print(Functions f, params int[] a)
{
  StringBuilder builder = new StringBuilder();
  builder.Append("f(");
  for (int i = 0; i < a.Length; ++i)
  {
    builder.Append(a[i]);
    if (i < a.Length - 1) builder.Append(", ");
  }
  builder.Append(") = ");
  builder.Append(f(a));
  Console.WriteLine(builder.ToString());
}
```

where the following delegates are defined in the start of the program:

```
public delegate int Function3(int a, int b, int c);
public delegate int Function0();
public delegate int Functions(params int[] a);
```



The statement

```
Print((a, b, c) => a + b + c, 11, 13, 17);
```

calls a method *Print()* with 4 parameters, the first actual parameter being a lambda expression, while the last three are integers. The lambda expression has three parameters. Note that they are in parentheses, which is necessary when there are several parameters. The expression performs a calculation whose result is an *int*, and the compiler can therefore translate the expression into a method defined by the delegate *Function3* and thus decide which *Print()* method to execute.

The following calls the same *Print()* method that prints the least of three numbers:

```
Print((a, b, c) => { int t = a < b ? a : b;
return t < c ? t : c; }, 17, 11, 13);
```

This time, note that the expression consists of a statement block with two statements. Note that the first statement creates a local variable that is local to the block that defines the lambda expression.

The next lambda expression is simple:

```
Print(() => rand.Next(1000));
```

Here, the parameter list for the expression is empty, which you specify with empty brackets. The value of the expression is an *int*, so the compiler can see that the lambda expression must be translated to the type delegate *Function0* and call the corresponding *Print()* method.

The following expression defines the type of the parameter as an *int* array:

```
Print((int[] a) => (a[0] < a[1] ? a[1] : a[0]), 23, 29);
```

Note that if the parameter for a lambda expression is specified with a type, the parameter must be set in parentheses. In this case, the expression defines a method that returns the largest, of the first two numbers in the parameter array. Since the expression parameter is an array, the compiler can see that the expression must be translated to an anonymous delegate of the type *Functions* and thus call the correct *Print()* method.

You can also use lambda expressions to define methods, and the compiler will create the methods from the lambda expressions:

```
class Calc
{
  public static int Add(int a, int b) => a + b;
  public static int Sub(int a, int b) => a - b;
  public static int Mul(int a, int b) => a * b;
  public static int Div(int a, int b) => a / b;
  public static int Mod(int a, int b) => a % b;
}
```

It is only a short syntax, and the compiler creates the same class as usual.

As the last example of lambda expression, I will show how to define an event handler with a lambda expression that is perhaps one of the most important uses. The following class run a method using an asynchronous delegate where the method runs in an infinity loop. For each iteration the loop sleeps random for 0 to one second, and then it generates a random number between 0 and 100. If the number is a prime and the method sends a notification to any observers:

```
class Generator
{
  public event NumberObserver Observers;

  public Generator()
  {
    (new Action(Run)).BeginInvoke(null, null);
  }

  private void Run()
  {
    while (true)
    {
      Thread.Sleep(Program.rand.Next(1000));
      int t = Program.rand.Next(100);
      if (Observers != null && IsPrime(t))
      Observers(this, new NumberArg(t));
    }
  }

  private bool IsPrime(int n) { ... }
}
```

The event has the type:

```
delegate void NumberObserver(object sender, NumberArg e);
```

where

```
class NumberArg : EventArgs
{
  public int Number { get; private set; }

  public NumberArg(int number)
  {
    Number = number;
  }
}
```

If you run the method

```
public static void Test07()
{
  (new Generator()).Observers += (sender, e)
  => Console.WriteLine(e.Number);
  while (true) Thread.Sleep(100);
}
```

it creates a *Generator* object and add an event handler for this object. Here you must note that the event handler is defined by a lambda expression, and it makes it very simple to define simple event handlers. The result is that the method within some interval prints a prime number on the screen.

You should especially note the last while statement where the purpose alone is that the method should not terminate. It is a simple, by no means optimal solution, a solution called *busy wait*.

8 OPERATOR OVERRIDING

Programming languages such as C# has a number of operators that work on the built-in types. Important examples are the calculation operators, which work on a number types such as *int*, *double*, etc., and other examples are the comparison operators. For example, if looking at addition, you can write something like

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

which simply means that the sum of the variables *b* and *c* is stored in the variable *a*. An operator is, in principle, just a method that, for addition, has 2 parameters and one could have used the syntax instead

a = +(b, c);

reflecting, that this is a method named +. However, the normal operator syntax is used, as it is standard for mathematical expressions, and it is simply a matter, that the meaning of the + symbol being built into the compiler. With compiler terminology, + is a token in the language C#.

How the compiler interprets a + depends on the types of values on which it acts. + means something different - does something different - depending on whether the type is *int* or *double*, and if the type is *string* instead, + means something completely different. Stated slightly differently and with the usual method terminology, the + operator is overridden.

Since the operators are built into the compiler, you can generally only apply them to the built-in types, and it does not make sense to perform the above addition on variables whose type is *Product*, *Person* or another custom type, but some languages including C# allow to some extent to override operators (so that they make sense) for custom types.

There are several limitations, but the three most important are:

- not all operators can be overridden
- it is not possible to define your own operators, that is new symbols for operators
- it is not always a good idea to use operator overriding

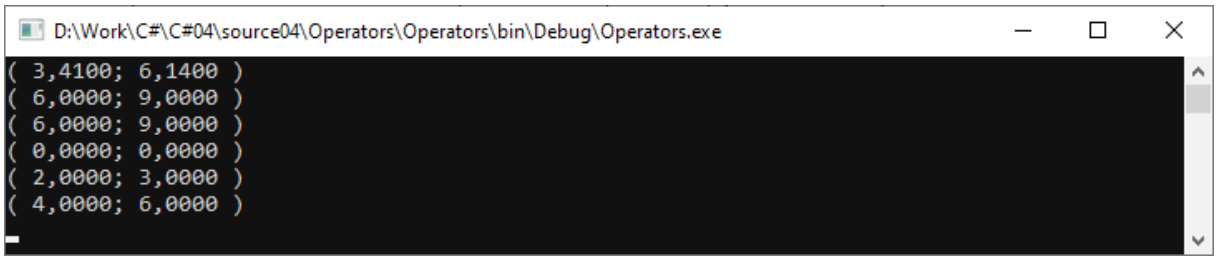
It should be added right away that C# is not as flexible in terms of operator overriding as one might wish, but conversely it is an option worth knowing.

As an example, I would write a class that defines a point consisting of two coordinates and overrides three operators. In addition, there is a test method where you should mainly notice how the operators are used. The class is written as:

```
public class Point
{
  public double X { get; set; }
  public double Y { get; set; }

  public Point()
  {
  }

  public Point(double x, double y)
  {
    X = x;
    Y = y;
  }

  public static Point operator +(Point p1, Point p2)
  {
    return new Point(p1.X + p2.X, p1.Y + p2.Y);
  }

  public static Point operator *(Point p, double t)
  {
    return new Point(p.X * t, p.Y * t);
  }

  public static Point operator *(double t, Point p)
  {
    return new Point(p.X * t, p.Y * t);
  }

  public override string ToString()
  {
    return string.Format("( {0:F4}; {1:F4} )", X, Y);
  }
}
```

and the test method is:

```
static void Test2()
{
  Point p1 = new Point(1.41, 3.14);
  Point p2 = new Point(2, 3);
  Point p3 = new Point();
  Console.WriteLine(p1 + p2);
  Console.WriteLine(p2 * 3);
  Console.WriteLine(3 * p2);
  Console.WriteLine(p3);
  p3 += p2;
  Console.WriteLine(p3);
  p3 *= 2;
  Console.WriteLine(p3);
}
```



Note that the two coordinates are defined using auto-generated properties as well as the class has a *ToString()* method. In addition, the class has an operator override of both the + and * operator. The first creates a new point initialized with the sum of the coordinates of two other points, while the second creates a point initialized with the coordinates of a second point multiplied by a constant (scalar multiplication). The last override is available in two versions where the difference is that the two parameters have been switched sequentially. For the override of operators, the usual rules apply to override methods.

Note how to define an operator override:

- an operator override is always a static method
- one of the parameters must have the same type as the class that defines the operator - usual the first parameter

Those are almost the only requirements. However, you should note that an operator override is static, which limits its use to some extent.

In the test program, you must note the fourth statements that applies the + operator to two *Point* objects. Also note the eighth statement where the += operator is used. For example, if you override the + operator, you have also automatically overridden the += operator, and this applies equally to all other operators that can be combined with the assignment operator. However, one requirement is that the operator (here +) has an override, where the type of the first parameter being the class that defines the operator (here *Point*). For example, on the override of * above, there must be an override where the first parameter has the type *Point* in order to write

```
p2 *= 2;
```

The following operators can be overridden:

- Unary operators: + - ! ~ ++ -- true false
- Binary operators: + - * / % & | ^ << >>
- comparison operators: == |= < <= > >=

However, for comparison operators, they must be overridden in pairs, that is < and <=. All operators in the binary operators list will also automatically override the operator combined with the assignment operator:

- += -= *= /= %= &= |= <<= >>=

8.1 THE CLASS COUNTER

As another example, I want to show a class that represents a simple counter that can count from 0 to an upper limit. The goal of the example is to show multiple operator overrides. The class is written as:

```
public class Counter
{
  public uint Size { get; private set; }
  public uint count = 0;

  public Counter(uint size)
  {
    Size = size;
  }

  private Counter(uint size, uint count)
  {
    Size = size;
    this.count = count;
  }

  public uint Count
  {
    get { return count; }
  }

  public static Counter operator ++(Counter c)
  {
    if (c.count >= c.Size) throw new
    ApplicationException("Operator overflow...");
    return new Counter(c.Size, c.count + 1);
  }

  public static Counter operator --(Counter c)
  {
    if (c.count == 0) throw new
    ApplicationException("Operator underflow...");
    return new Counter(c.Size, c.count - 1);
  }

  public static bool operator ==(Counter c1, Counter c2)
  {
    return c1.count == c2.count;
  }
```

```
  public static bool operator !=(Counter c1, Counter c2)
  {
    return c1.count != c2.count;
  }

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

  public override int GetHashCode()
  {
    return count.GetHashCode();
  }

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

and the test method is:

```
static void Test2()
{
  Counter c1 = new Counter(10);
  Counter c2 = new Counter(10);
  Console.WriteLine(c1);
  for (int i = 0; i < 5; ++i) ++c1;
  Console.WriteLine(c1);
  while (c2 != c1) ++c2;
  Console.WriteLine(c2);
  Counter c3 = ++c1;
  Counter c4 = c2++;
  Console.WriteLine(c1);
  Console.WriteLine(c2);
  Console.WriteLine(c3);
  Console.WriteLine(c4);
  while (c1.count > 0) --c1;
  Console.WriteLine(c1);
}
```

The Counter class should show how to implement the comparison operators, as well as the two unary operators ++ and --.

The class is, as mentioned, a simple counter that can count from 0 up to a value represented by Size. The implementation of the comparison operators is trivial and you only need to note the syntax. In this case, I override ==, and if you do, the compiler demands that you override !=. Also note that if you override == then you get a warning which recommends that you also override *Equals()* and *GetHashCode()*. Above, for example, I have overridden two of the comparison operators, namely equal to and different from, and usually the other comparison operators will also be overridden.

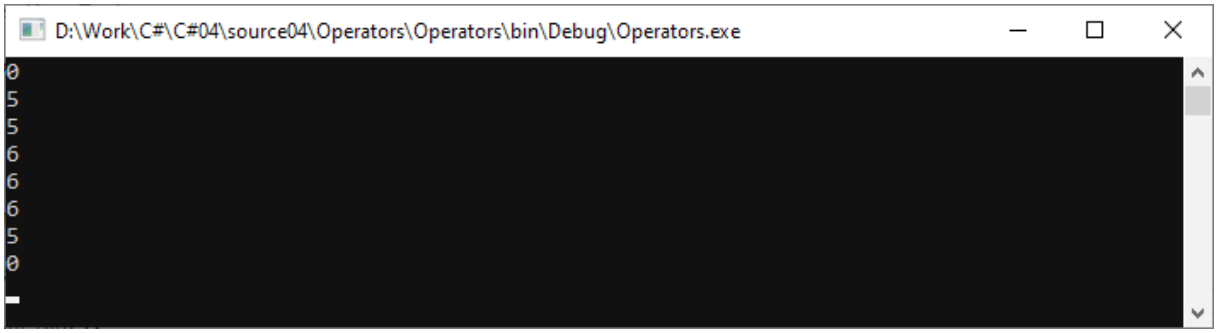
The class also overrides the ++ operator, and the code does not require much explanation, but the effect is not obvious. Note, however, that the code does not appear to change the value of the parameter c, but it does occur implicitly. First, note that you have two versions of the ++ operator:

```
++i
i++
```

and for the usual native types, the version matters if the operation is included in an expression. In C#, it is only possible to override ++ in one way, but in return, the usual effect is mysteriously obtained.

In the test program, two *Counter* objects are created. Note the *for* loop that counts the first object up 5 times, thus changing the value of objects *c1*. The same goes for the first *while* loop, but here it is instead *c2* that counts. Note the condition in the *while* loop, where it is utilized that the class *Counter* overrides the operator different from.

Then note how *c3* is initialized: *c1* is counted and the value is assigned to *c3*. *c4* is initialized with *c2*, which is then counted.



EXERCISE 10: A POINT TYPE

Create a console application project which you can call *PointProgram*. Add the same type *Point* as in the program *Operators* to you project to make the following changes:

1. You must change the type from a *class* to a *struct*
2. The type must override the comparison operators == and !=
3. You must add another overriding of the * operator, when it must have two *Point* parameters and return and *double* that is the length between the two points

Remember to test the new operators from *Main()*.

EXERCISE 11: A TIME TYPE

Create a new WPF project. Add a struct called *Time* when the struct must represent a time a day. The time must be represented as an *int* where the value is the number of milliseconds from the start of the day. Note that this means that the value of a *Time* object will be within 0 and 86399999 (there is 86400000 milliseconds a day). The struct should not have any public constructor, and that means that a new *Time* has the value 0.

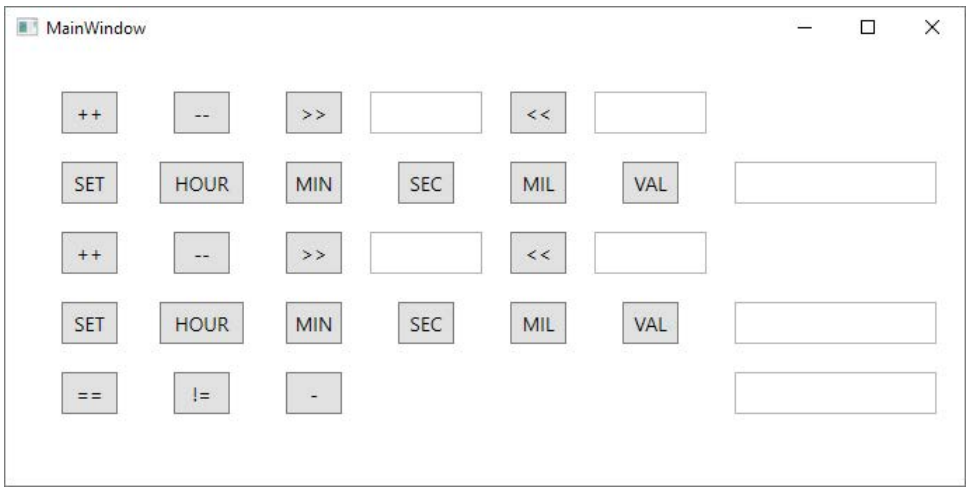
The struct must have the following read-only properties:

1. *Hour* which returns the hour
2. *Min* which returns the minutes
3. *Sec* which returns the seconds
4. *Milli* which returns the milliseconds
5. *Value* which returns the internal value

The struct must also have a method *Set()* which initialize an object reading the hardware clock. As the last the struct must override the following operators:

```
// the number of milliseconds between t1 and t2 as signed int
public static operator-(Time t1, Time t2)

// Increment operator which increase the time 1 millisecond
public static Time operator ++(Time t)

// Decrement operator which decrease the time 1 millisecond
public static Time operator --(Time t)

// Add m milliseconds to the time t, may be the time must turn around
public static Time operator >>(Time t, int m)

// Subtract m milliseconds from the time t,
may be the time must turn around
public static Time operator <<(Time t, int m)

// Comparison operators
public static bool operator==(Time t1, Time t2)
public static bool operator !=(Time t1, Time t2)
```

When you have write the struct you must test it using the window below. The three *TextBox* components to the right are for results. The program must have two *Time* objects, and the first two lines of components should be used to test the first *Time* object, while the two next lines of components should be used to test the other *Time* component. The last three buttons should be used to compare the two *Time* objects.



8.2 THE INDEXER

I have mentioned and used the indexer before, and in reality there is no more to tell, but I will briefly refer to it again as it is also a form of operator override.

If you look at an array you refer to the individual elements with the array name and an index, for example

```
array[7] = 23;
```

a basic notation used in almost all programming languages. In some languages and also C# you can use this notation for other collections than arrays, and as shown in the chapter about collection classes many of the collection classes support this notation, and you can also use the notation for your own collection classes or classes in general. You do this by overriding the index operator [] with a slightly special notation, which is called an *indexer* and the code is implemented as a property with a special syntax. The index operator is thus overridden in a slightly different way than the other operators. To recall how I will show an example with a class *Zipcodes* which is a collection with *Zipcode* objects:

```
class Zipcode
{
  public string Code { get; private set; }
  public string City { get; private set; }

  public Zipcode(string code, string city)
  {
    Code = code;
    City = city;
  }

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

It is a very simple class that requires no special explanation. The class *Zipcodes* reads the content of a text file with Danish zip codes and creates a collection with *Zipcode* objects based on this:

```
class Zipcodes : IEnumerable<Zipcode>
{
  private IDictionary<int, Zipcode> table = new
  SortedDictionary<int, Zipcode>();

  public Zipcodes()
  {
    LoadData();
  }

  public Zipcode this[int code]
  {
    get
    {
      try
      {
        return table[code];
      }
      catch
      {
        return null;
      }
    }
  }

  public Zipcode this[string city]
  {
    get
    {
      city = city.ToLower();
      foreach (Zipcode zipcode in table.Values)
        if (zipcode.City.ToLower().Equals(city)) return zipcode;
      return null;
    }
  }

  public IEnumerator<Zipcode> GetEnumerator()
  {
    foreach (int code in table.Keys) yield return table[code];
  }
```

```
  System.Collections.IEnumerator System.
  Collections.IEnumerable.GetEnumerator()
  {
    return this.GetEnumerator();
  }

  private void LoadData()
  {
    StreamReader file = new StreamReader("zipcodes");
    for (string line = file.ReadLine(); line != null; line = file.ReadLine())
      ParseLine(line);
    file.Close();
  }

  private void ParseLine(string line)
  {
    try
    {
      string[] elems = line.Split(';');
      if (elems.Length == 2)
        table.Add(int.Parse(elems[0]), new Zipcode(elems[0], elems[1]));
    }
    catch
    {
      Console.WriteLine(line);
    }
  }
}
```

I should not mention how to read the file, but the method *LoadData()* is called from the constructor and it initialize a dictionary with *Zipcode* objects where the code is used as a key of the type *int* while the value is a *Zipcode* object. The class defines an indexer:

```
public Zipcode this[int code]
{
  get
  {
    try
    {
      return table[code];
    }
    catch
    {
      return null;
    }
  }
}
```

It is a property named *this*, and in this case it is a read-only property, but an indexer can also have a *set* part. The index is an *int* and it returns the *Zipcode* for that code. It is simple as a dictionary also has an indexer for the key. When this indexer throws an exception if the key does not exists the reference is placed in a *try* / *catch*, and if the key is not found the indexer returns *null*. You must note the syntax and the use of the work *this*.

An indexer is a property which depends on an index, here an *int*, but it can be overridden like other methods. In this case the indexer is overridden for a *string* which represents the *city* name, but you must note that the indexer this time works using search and as so the indexer has a poor performance, and the most important justification for this index is also to show that an indexer can be overridden.

The class also implements the iterator pattern. The goal is to show the application of the *yield return* statement. It is a statement used to automatic create an iterator where you loop over a collection of objects and each *yield return* statement returns the next object.

9 USER DEFINED TYPE CONVERTING

If you have two variables *a* and *b*, you cannot generally write

```
a = b;
```

since the statement is not necessarily meaningful - the types must be right. For example, if *a* is a *long* and *b* is an *int*, the statement can be executed immediately, as one can easily copy an *int* to a *long* - an implicit type cast occurs, but is the other way around where *a* is an *int* and *b* is a *long*, you get an error as you cannot necessarily copy a *long* to an *int*. Here's an explicit type of cast is needed:

```
a = (int)b;
```

However, it is not always possible to type cast one type into another type. For example, if *a* is an *int* and *b* is a *string*, then you cannot copy *b* to *a* by writing a type of cast - quite reasonable, because a *string* generally cannot be converted to an *int* in a sensible way.

C# has a type *System.Drawing.PointF* that similarly to the type *Point* above represents as a point in a coordinate system, but the coordinates of a *PointF* have the type float. The following statements are therefore legal:

```
System.Drawing.PointF p1 = new System.Drawing.PointF(2.72F, 1.73F);
Point p2 = new Point(3.14, 1.42);
```

but if you then try to write

```
p2 = p1;
```

you get an error, because the compiler does not know how to convert a *System.Drawing.PointF* to a *Point*, although this should be possible, since it is simply a matter of converting two *float* coordinates to two *double* coordinates. To resolve this issue, expand the class *Point* with a custom type of cast:

```
public static implicit operator Point(PointF p)
{
  return new Point(p.X, p.Y);
}
```

Then it is possible to assign *p1* to *p2*, because now the compiler knows how to convert a *PointF* object to a *Point*. The type cast is defined implicitly so that one does not need to specify a type cast in the code. Below is a similar custom type cast to convert a *Point* to a *PointF* (that is a conversion the other way):

```
public static explicit operator PointF(Point p)
{
  return new PointF((float)p.X, (float)p.Y);
}
```

Then you can also write the following statement:

```
p1 = (PointF)p2;
```

but this time an explicit type cast is needed as the custom type cast is explicitly defined.

A custom type of cast is actually a form of operator override (the operator ()). Note the syntax:

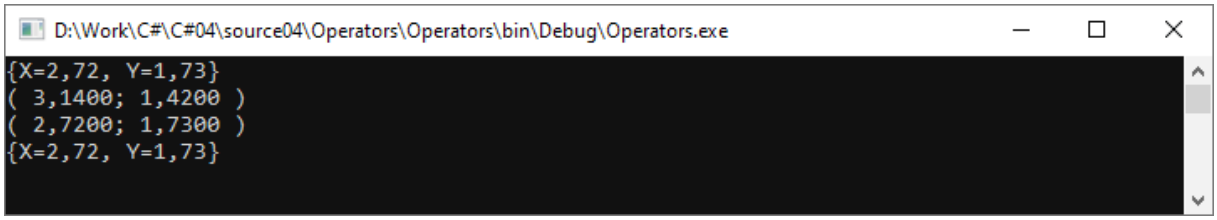
operator Type1(Type2 t)

where *Type1* is the type to convert to while *Type2* is the type to convert from.

If you write a class, it is up to the programmer to define the type casts that should be possible. In addition, for each custom type of cast, one must decide whether it should be implicit or explicit, and of course one should not define custom type casts as implicit unless it makes good sense.

The following method uses the above type casts:

```
static void Main(string[] args)
{
  System.Drawing.PointF p1 = new System.Drawing.PointF(2.72F, 1.73F);
  Point p2 = new Point(3.14, 1.42);
  Console.WriteLine(p1);
  Console.WriteLine(p2);
  p2 = p1;
  Console.WriteLine(p2);
  p1 = (System.Drawing.PointF)p2;
  Console.WriteLine(p1);
}
```



10 FINAL EXAMPLE: A CONVERTER

In everyday life, we measure a wide range of different sizes, such as temperature, distances, weights, etc., and common to these sizes is that we often indicate results in different units. For temperature, for example, it can be Celsius or Fahrenheit. There is often a need to be able to convert from one unit to another. As the final example in this book, I will look at the development of a program that can perform such conversions.

The final example in the previous book was a calculator, and here was the biggest challenge to implement algorithms that could parse and evaluate an expression. With that in place, it was in principle simple to write the code. This corresponds to the fact that the development had an emphasis on design. The current program is also a simple program and even simpler than the previous program, but in this program the emphasis is on analysis, and thus to determine what the program should be able to do. The development of the program is carried out through the following iterations:

1. Analysis
2. Development of a prototype
3. Design of the architecture and model
4. Programming
5. Code review and the last thing

10.1 ANALYSIS

The program must be able to perform typical unit conversions that must be done within certain categories such as temperature, length, volume and so on. Typically, you enter a number within a category, select the unit and which unit the number should be converted to. Here it must be possible to use a prefix for the number, where it makes sense:

Yotta	Quadrillion	Y	10^{24}
Zetta	Trilliard	Z	10^{21}
Exa	Trillion	E	10^{18}
Peta	Billiard	P	10^{15}
Tera	Billion	T	10^{12}
Giga	Milliard	G	10^{9}
Mega	Million	M	10^{6}
Kilo	Thousand	k	10^{3}
Hekto	Hundred	h	10^{2}
Deka	Ten	da	10
Deci	Tenth	d	10^{-1}
Centi	Hundredth	c	10^{-2}
Milli	Thousandth	m	10^{-3}
Micro	Millionth	μ	10^{-6}
Nano	Milliardth	n	10^{-9}
Piko	Billionth	p	10^{-12}
Femto	Billiardth	f	10^{-15}
Atto	Trillionth	a	10^{-18}
Zepto	Trilliardth	z	10^{-21}
Yokte	Quadrillionth	y	10^{-24}

The following conversions are currently planned:

Numbers (integers)

- Decimal
- Hexadecimal
- Binary
- Roman

Volume

- Cubic fathom
- Cubic mile
- Cubic foot
- Cubic yard
- Cubic inch
- Barrel (Imperial)
- Gallon (Imperial)
- Barrel (US)
- Gallon (US)
- Hectoliters
- Cubic meters
- Cubic kilometers
- Cubic centimeter
- Cubic millimeter
- Cubic nano meter
- Cubic micrometer
- Liter
- Milliliter
- Micro liters
- Centiliter
- Deciliter
- Pint (Imperial)
- Quart (Imperial)
- Pint (US)
- Quart (US)

Distance

- Astronomical unit
- Meter
- Foot
- Inch
- Light years
- Mile
- Nautical Miles
- Parsec
- Yard

- Cubit
- Fathom
- Furlong

Weight

- Ton
- Kilogram
- Pound
- Ounce
- Gram

Temperature

- Celsius
- Fahrenheit
- Kelvin

Area

- Acre
- Hectare
- Square Kilometers
- Square Centimeters
- Square millimeter
- Square meter
- Square mile
- Square foot
- Square inch
- Square yard
- Barn

Angle

- Degree
- Radian

Time

- Atomic time unit
- Day
- Hour
- Minute
- Second
- Month
- Week
- Year

There are many other units, both within each category and, and on the other hand, many other categories, especially categories relating to physics units. You also need to be aware that the size of a unit in several cases is geographically determined, for example, there may be a difference between the European and the American standard.

In order to convert from one unit to another you must have a formula, and that is, you has to go back to the definition of the individual units. In the vast majority of cases, the formula consists merely of a factor where the value in one unit is to be multiplied by a value. For example, if you has to convert a value in kilograms to a value in grams, then simply multiply the value by 1000. In most cases, the work therefore consists of determining this factor.

However, with reference to the above categories, there are two exceptions. One is Number, which is not really a conversion, but rather a question of representing the same number in several ways. The conversion between decimal, hex decimal and binary is trivial, as C# has the necessary tools. With regard to Roman numerals, one has to have algorithms that can convert between decimal and Roman numerals. In this program, I will apply the following definition of a Roman numeral.

I = 1
V = 5
X = 10
L = 50
C = 100
D = 500
M = 1000

A Roman numeral must be written as short as possible using the following rules:

1. If a smaller number is written before a larger number, the smaller number is subtracted
2. If a smaller number is written after a large number, the smaller number is added
3. The largest number should be on the left if it is not to be subtracted
4. I, X, C and M must be added one, two or three times, and must then stand together
5. I, X and C must only be subtracted once (I can only stand in front of V and X, X can only stand in front of L and C, C can only stand in front of D and M)
6. V, L and D should only be used once

According to these rules, the largest possible Roman numeral is: MMMCMXCIX = 3999.

The second category where the conversion cannot be performed directly by multiplying by a factor is Temperature. Here are 6 formulas needed:

1. Fahrenheit = Celsius * 1.8 + 32, Celsius >= -273.15
2. Kelvin = Celsius + 273.15, Celsius >= -273.15
3. Celsius = (Fahrenheit - 32) / 1.8, Fahrenheit >= -459.67
4. Kelvin = (Fahrenheit - 32) / 1.8 + 273.15, Fahrenheit >= -459.67
5. Celsius = Kelvin - 273.15, Kelvin >= 0
6. Fahrenheit = (Kelvin - 273.15) * 1.8 + 32, Kelvin >= 0

All the other conversions are made using a factor, and the work consists of determining these conversion factors (there are 650). I have inserted the values into an XML document where the start is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <category text="Angle" pref="0">
    <rules text="Degree">
      <rule text="Radian" value="0.017453292519943"/>
    </rules>
    <rules text="Radian">
      <rule text="Degree" value="57.295779513082320"/>
    </rules>
  </category>
  <category text="Volume" pref="1">
    <rules text="Cubic fathom">
      <rule text="Cubic mile" value="0.0000000000067936"/>
      <rule text="Cubic foot" value="216"/>
      <rule text="Cubic yard" value="8"/>
      <rule text="Cubic inch" value="373248"/>
```

The reason for this document is that you can then write a program that dynamically builds the user interface by reading this document, and thus you from the program can edit the and expands with new conversion roles. Of course, one must be aware that if you change the document, it must be syntactically correct and contain legal values.

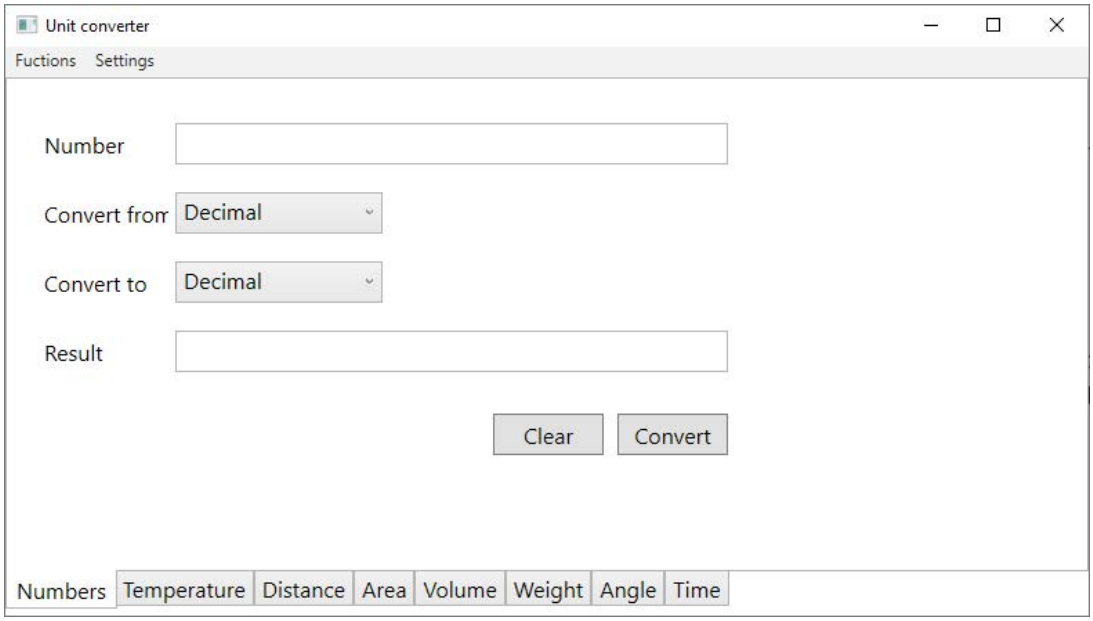
The program should have a relatively simple user interface which mainly consists of two windows. The main window is used for conversions and is primarily a matter of the user being able to choose the unit category and which units to convert between. The second window should be used to maintain the XML file with conversion factors. Here it must be possible to maintain existing rules, but it must also be possible to create both new categories and rules within the categories and store the rules in a file selected by the user.

10.2 THE PROTOTYPE

To define the user interface I have written a simple prototype, which is a Visual Studio WPF project called *Converter*. The prototype consists of two windows:

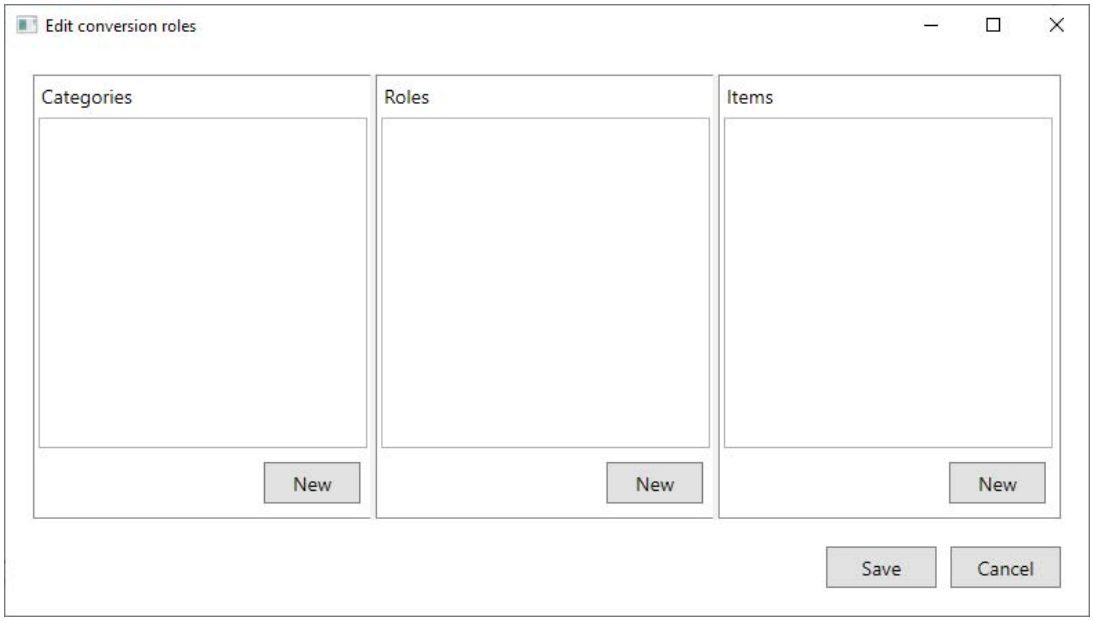
1. *MainWindow*
2. *EditWindow*

where the first is the converter and the other is used for edit the conversion factors and the XML document. When the program starts it should open a window which look likes the following:



The window consists of tabs with one tab for each category. In the prototype only the first two has a content because the others must be created dynamic in code as there content are determined by the XML document.

The window also has a menu and for moment it is not decided what menu items the menu should have, but under *Functions* it is a menu item that opens the window for maintenance conversion rules:



The window has three list boxes for categories, rules for the selected category and items for the selected rule.

10.3 DESIGN

The design includes two things that are partly the model and partly the loading of the XML document. The starting point for the design is to copy the project with the prototype (the copy is called *Converter0*) and then modify the original project.

For the model, a class *Rule* is defined which represents a conversion rule from one unit within a category to another entity within the same category. Thus, with reference to the XML document, the class represents a *rule* element.

```
namespace Converter.Models
{
  class Rule
  {
    public string Name { get; set; }
    public double Factor { get; set; }

    public Rule(string name, double factor)
    {
      Name = name;
      Factor = factor;
    }

  public double Convert(double value)
  {
    return Factor * value;
  }
  public override string ToString()
  {
    return Name;
    }
  }
}
```

Here, *Name* represents the name of the unit to convert to, and the name should only be used to display the rule in the user interface. The class *Rules* is used to represent all the units that a particular unit within a category can be converted to:

```
namespace Converter.Models
{
  class Rules : IEnumerable<Rule>
  {
    public string Name { get; set; }
    private List<Rule> list = new List<Rule>();

    ...
  }
}
```

and again the field *Name* if used for the value shown in the user interface. The class *Category* represents a category of units and has a collection of *Rules* objects:

```
namespace Converter.Models
{
  class Category : IEnumerable<Rules>
  {
    public string Name { get; set; }
    private List<Rules> list = new List<Rules>();
    ...
  }
}
```

Finally, there is the program's model, which is quite simple and is only a collection of *Category* objects, as well as a variable that contains the number of decimals for results:

```
namespace Converter.Models
{
  class MainModel : IEnumerable<Category>
  {
    public int Dec { get; set; }
    private List<Category> categories;

    public MainModel(List<Category> categories)
    {
      this.categories = categories;
    }

    public IEnumerator<Category> GetEnumerator()
    {
      return categories.GetEnumerator();
    }

    System.Collections.IEnumerator System.
    Collections.IEnumerable.GetEnumerator()
    {
      return this.GetEnumerator();
    }
  }
}
```

The list with *Category* objects should be initialized in the constructor, and the list should be created, when the program starts. At that time the program must read the XML document and parse it to build the model, and it means two problems must be solved:

1. Where to save the XML document
2. How to parse a XML document

For the first, it has been decided that the XML document should be saved in a directory

```
C:\Torus\Converter
```

but it must also be possible for the user to load and store the file somewhere else in the filesystem. This means that the user can select which conversion rules to use.

To make sure it does not trouble reading the XML file, I have written a small test program that reads and parses the file, which is quite simple as the framework has the necessary classes for that purpose. You should note that it is quite common for programmers during the design phase to write these kind of test programs in order to test a technology or perhaps test an algorithm. The aim is to identify uncertainties as early as possible, uncertainties that may present challenges later during programming. In this case, I have written the following test program:

```
static void Main(string[] args)
{
  using (XmlReader reader = XmlReader.Create("UnitsDefinitions.xml"))
  {
    while (reader.Read())
    {
      if (reader.IsStartElement())
      {
        Console.WriteLine("Element: " + reader.Name);
        while (reader.MoveToNextAttribute())
          Console.WriteLine(" " + reader.Name + " = " + reader.Value);
      }
    }
  }
}
```

An *XmlReader* is a class representing an XML document. An object opens the document and you can then read the document element for element through a one way parse of the document. The statement *reader.Read()* read the next element until there not are more elements and for each read set an internal cursor to the current node. For each element in the XML tree *reader* has some methods and here methods to iterates the attributes for the current node. The result is that the above program reads all elements in the document, print the element's name and all the attributes.

The .NET framework also has a class *XmlWriter* that can be used to write an object in a file as an XML docuemnt.

10.4 PROGRAMMING

I want to split the programming in two iterations:

1. Conversion and then the main program.
2. Maintain of conversion rules.

The rationale for this split is that you can quickly get a part of the program ready for testing by the future users, so that you are aware of any adjustments as early as possible. In this iteration I will program the main program and the the *MainWindow*:

1. Create a copy of the project from the design. I have called to copy *Converter1*.
2. Add two simple enumerations to the model layer that defines names for temperature and numbers.
3. Add a user control called *NumbControl* to the view layer. The control should be used to convert numbers, and the XML can be copied from the prototype. After creating the control the code for the first tab in the prototype is removed and a new tab using the above user control is added in code behind.
4. Write a controller *NumbCtrl* for the user control (a class in the controller layer) when the controller should have one method to perform a conversion:

```
public void Convert(string value, Numbers from, Numbers to)
```

   The method must raise an event with the converted value as argument, when a conversion is performed, and the method should also raises an event if the conversion could not be performed. The view (that is the user control) should be observer for these events.

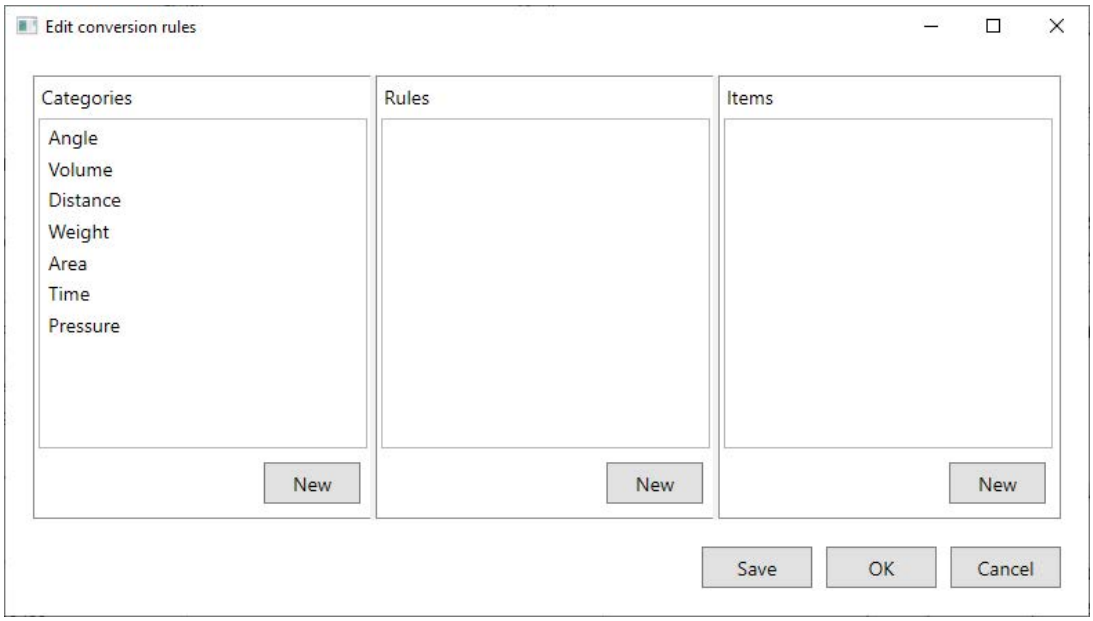
5. The Numbers tab is now implemented as a user control. As the next step I do the same for the Temperature tab and then do the same as in step 2) and 3) and create a user control *TempControl* and a controller class *TempCtrl*.
6. Read the configuration file *UnitsDefinitions.xml*. For now the file is copied to the program folder. The class *MainModel* is extended with a method which read and parses the file and instantiates a hierarchy of objects consisting of a collection

of *Category* objects as a collection of *Rules* objects and each *Rules* object as a collection of *Rule* objects. The method is called from the constructor and when a *MainModel* object is created (in code behind for *MainWindow*) the program has all the data it needs.

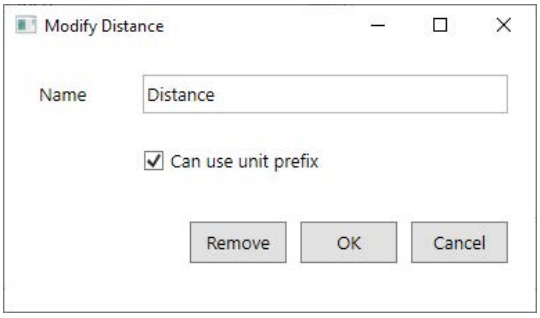
7. To the model layer is added a class *Prefix* representing a unit prefix. There is also added a class *Prefixes* representing all prefixes.
8. The next is a view class *CategoryControl* which is a user control for tabs for the dynamic created conversion rules. For this class is also created a controller class called *CategoryCtrl*. With these classes the code behind for *MainWindow* is modified to use the new user control and then the converter is ready for use. You should note that all the code in code behind used to dynamic creates the tabs is removed as it is no longer needed.

10.5 CONVERSION RULES MAINTENANCE

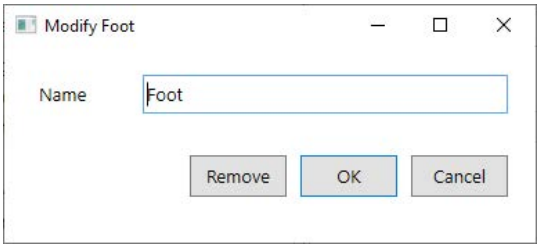
In this iteration I want to program the function to maintain the conversion rules, which also includes the ability to dynamically save the configuration and reload it. The iterations starts with creating a copy of the project called *Converter2*. When you from the menu select *Edit rules* you a get a window showing the categories:



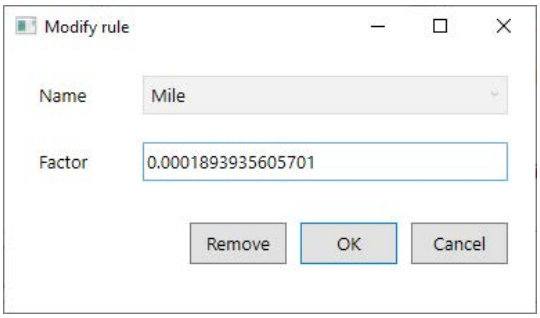
If you click on a category the program shows in the middle list box all rules defined for this category, and if you here click on a rule the program shows in the last list box all conversions rules for this rule. If you in one of the three list boxes double click on an item or clicks on *New* you get a dialog box where you can edit edit the item or create a new. If you for example double click on an item in the first list box the result could be:



and the same dialog box is used if you click *New* to create a new category. If you double click on an item in the middle list box you get the following dialog box



and a the dialog box to edit a conversion rule is:



To program this tool some functionality is needed:

1. To the controller layer is added a class *MainCtrl* with two functions called respectively *Save()* and *Load()* used when the user wants to save a configuration or load a saved configuration. The two methods are called from the menu while the first also is called from the window *EditWindow*. The methods raises an event to notify the *MainWindow* when program's configuration is changed.
2. The class *MainModel* is updated. The most important is a method that writes the configuration to an XML document. You can do this in several ways, but here is used an *XmlWriter*, which is a very simple and direct way. Of course, it is crucial that the document has the correct format.

3. There is added three new classes to the view layer that all represents windows, one for each of the three above dialog boxes. These are all simple dialog boxes and I have not added corresponding controller classes.
4. The class *EditWindow* (from the prototype) is changed with a new button, at button such you can keep the changes but without saving the configuration. Most changes relate to code behind where event handlers has been added for the user interaction.
5. Also the code behind for *MainWindow* is updated with event handlers for two menu items and an event handler as listener for the controller layer.

10.6 THE LAST THING

After programming, there is still something to do, and in this case the following must occur:

1. The menu must have a menu item with a function where you can set precision for the results. That is a dialog box (*PrecWindow*).
2. When you close the program it should save the current settings. That is the precision and the name for the file with the configuration, and these settings should be used the next time you run the program.
3. A code review.
4. Test.
5. Install the program.

The first task is quite simple and should have no more comments here.

For the second it is decided that the settings should be saved in a file *Converter.conf* in the directory

```
C:\Torus\Converter
```

and thus in the same directory as the directory with the configuration file. The file *Converter.conf* contains only one line with the precision and filename for the conversion rules separated by a semicolon. This file is created every time you terminate the program. The program read the file every time the program starts and create a new model corresponding to the content of the file. If the file does not exists the program creates a model with the default settings. All the program logic to save and load the settings are placed in *MainCtrl*.

The directory

```
C:\Torus\Converter
```

is hard-coded which is unfortunate and to help with that you can in Visual Studio insert a line in the configuration file *App.config*:

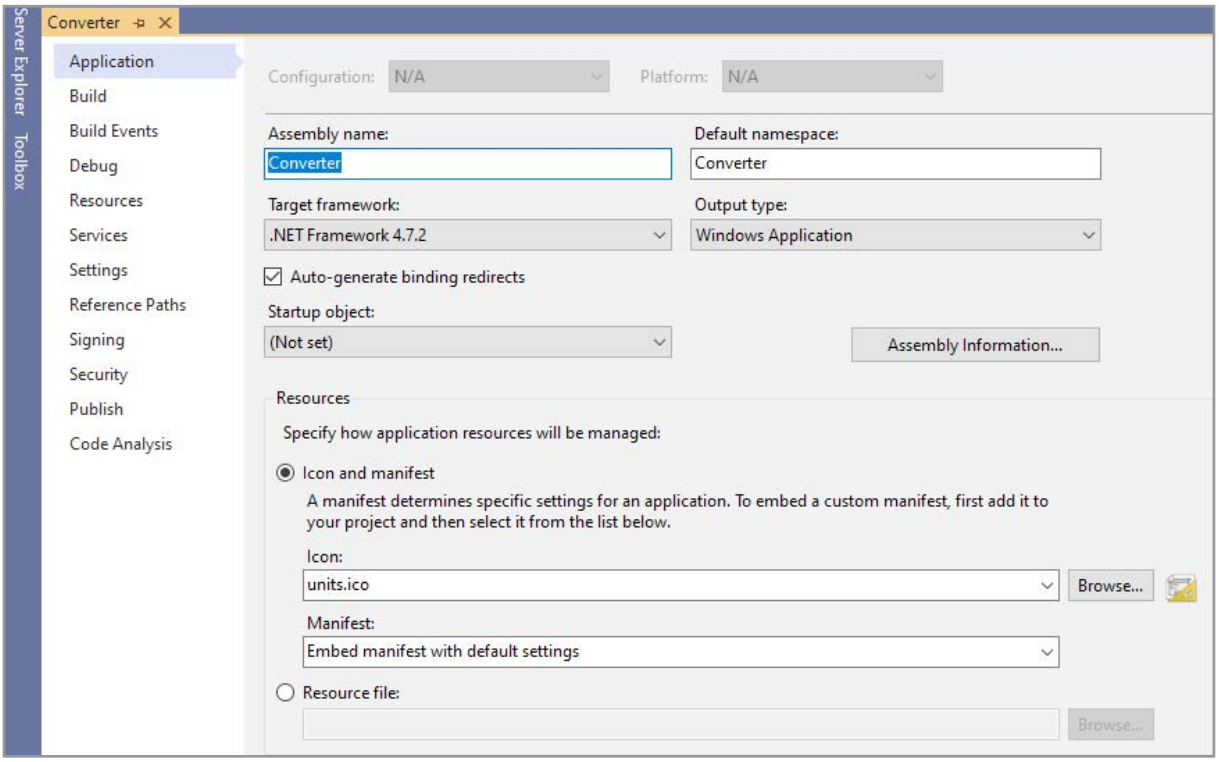
```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
  </startup>
  <appSettings>
    <add key="converter" value="C:\\Torus\\Converter\\" />
  </appSettings>
</configuration>
```

A user can then change the value if the files should be saved in another place. The value is loaded in *MainModel*:

```
public static string GetConfiguration()
{
  return ConfigurationManager.AppSettings.Get("converter");
}
```

The code review and test is performed in the same way and for the same reasons as in the final example in the previous book, but I will not in this place comment more on these two but very important activities.

To install the program I need an icon and as so a small image file converted to an *ico* file. To create such an icon you can on the Internet find a tool where you online for example can convert a png file to an ico file. When you have the icon you can add the icon to your program under properties:



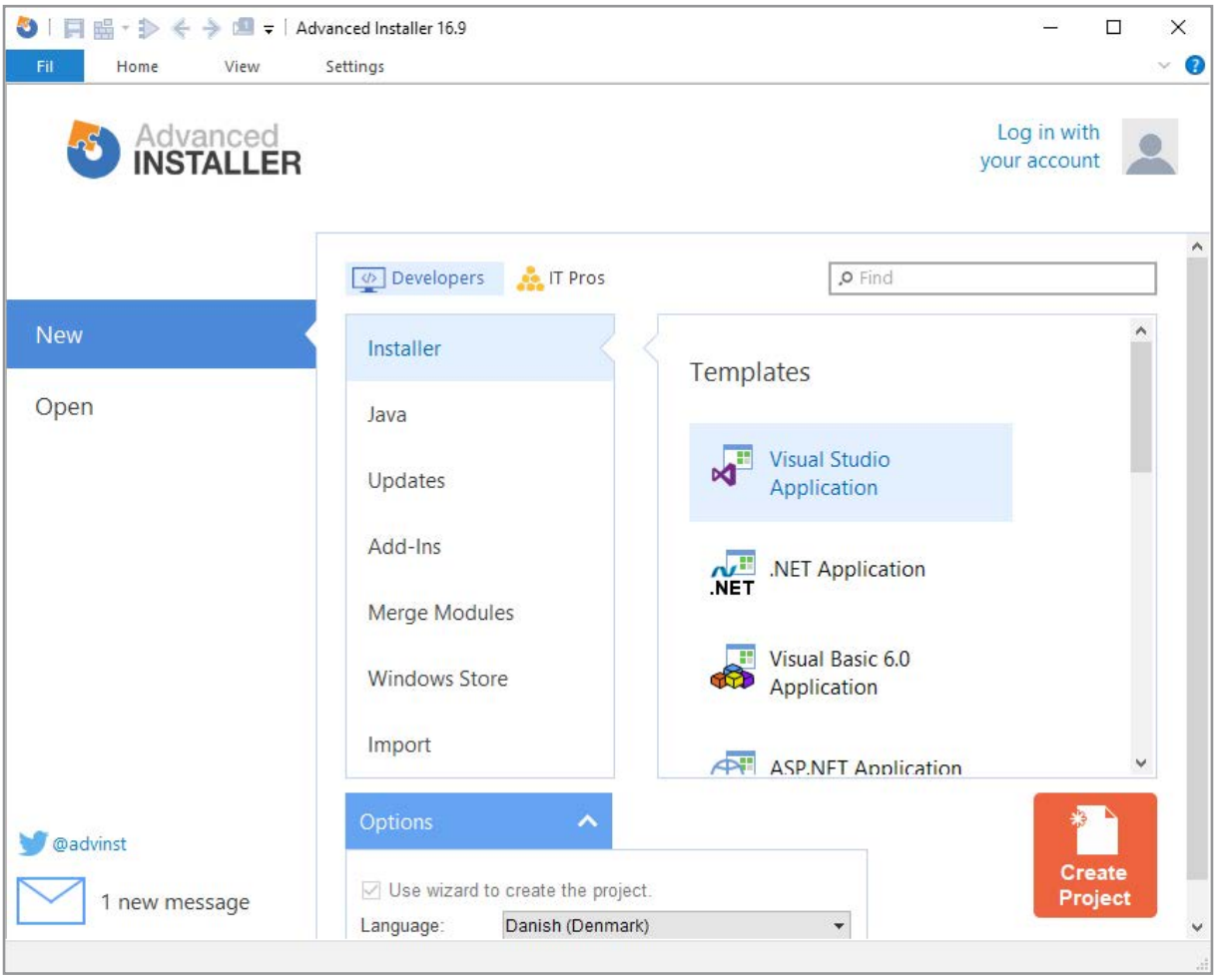
Before the program is ready for install on a computer you should build the project as a Release (choose Release in the menu).

Then there is the installation which means that the program must be copied to a directory in the *Program* folder, but also other files on which the program relies. At the same the program should be installed and added to the *Start* menu and maybe has an icon at the desktop. In this case only the files *Converter.exe* and *Converter.exe.config* must be copied, but it is also necessary to create the directory

```
C:\Torus\Converter
```

and copy the file *UnitsDefinitions.xml* to that directory.

Usually you create a MSI installer file which requires a tool that is not directly part of Visual Studio. One solution is to download and install a program *Advanced Installer* which can to some extent solve the task:



However, the full use of the product requires a license, but there is a template for a Visual Studio project that can be used without a license that easily builds an MSI file for the program. After that the program can be installed immediately, but you have to manually create the above directory and copy *UnitsDefinitions.xml* to it. The result is an installed program that can be used on the machine, and also a program that can be uninstalled again if want to.