Poul Klausen

C# 8

Threads and LINQ

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 8: THREADS AND LINQ

SOFTWARE DEVELOPMENT

C# 8: Threads and LINQ: Software Development
1st edition

ISBN 978-87-403-3536-1

CONTENTS

FOREWORD

This book is the eighth in a series of books on software development. The programming language is C#, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process, and how to develop good and robust applications. This book contains two subjects, which has nothing to do with each other and thus the book consists of two parts, which in principle can be read independently of each other. The first part of the book deals with multi-threaded programs and how to synchronize threads. The second part of the book deals with LINQ which primarily is extensions methods defined for collection classes. With these methods many operations on collections can be performed simpler and more efficient than writing the code yourself.

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

This book contains two subjects, which has nothing to do with each other and thus the book consists of two parts, which in principle can be read independently of each other. The first part of the book deals with multi-threaded programs while the second part deals with LINQ.

A thread is a method that can be formed independent of the rest of the program. Every program starts as least one thread, but by starting multiple threads it will appear to us users as the program performs several operations in parallel with each other, and that is exactly one of the reasons for allowing the program to start multiple threads. In principle, it is quite simple to create and start a thread, but in practice, multiple threads often have to use the same resources, and here problems easily arise as threads are executed independently of one another, and therefore one has no control over the order in which the threads uses a shared resource. This means that it is the programmer that it is responsible to synchronize threads use of shared resources, and this is by no means simple. Therefore, in practice, threads are always surrounded by some challenges.

Many tasks in programming is about to loop over objects in an array or collection and do something with the objects, for example to select a subset of objects which meets some criteria. All these tasks are always solved in the same way, and the result is that we as a programmer have to write almost the same code many times. LINQ (*Language Integrated Query*) is an extension of C#, which makes it easier to solve such tasks using a syntax extracted from SQL. In addition, LINQ can be integrated with database programming and make it much easier to access databases, and many even think of LINQ as an extension to database programming.

Before addressing the book's two main topics, I want to start with a very general introduction to .NET's process concept, primarily for following about threads.

1.1 A RUNNING PROGRAM

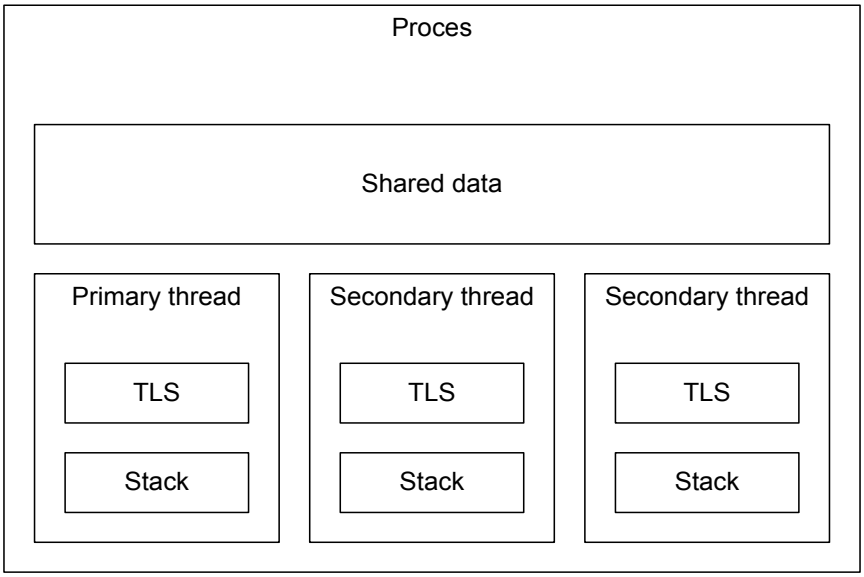
In the next chapter, I will as mentioned discusses how to write programs with multiple threads under the .NET platform, and in that context also what .NET makes available for thread synchronization. Before I look at threads, it is necessary to look at how a .NET program run as a process.

Viewed from Windows, a process is a definition of the resources that a running application uses, including the necessary memory allocation. Each time you start a program by loading the exe file into memory Windows creates a new process for running that program as well as allocates a memory area (an address space) for the program. At the same time, the process is assigned a unique process number called PID. If you want to know which processes are currently running on the machine, you can press Ctrl + Shift + Esc. It is thus far along the way to draw a similarity between a process and a running program.

Processes are isolated entities so that processes cannot immediately communicate with each other. A process has for example no access to data for another process, and that is important, because if a process fails and cannot continue, then it is only this process and the program that runs the process that stops, and not all other programs. If processes need to communicate - and sometimes they do, it is necessary to make use of special operating system mechanisms.

A thread is a specific sequence of instructions in a process whose execution are controlled by the operating system as a unified block of instructions, and a thread is usually started by executing a method. All processes has at least one thread, which is called the primary thread, which is started when the program is started - when the program's *Main()* method is executed, a process is created that starts the primary thread. Many processes - for example processes for small demo programs - has only one single thread and is therefore called single threaded programs. Such processes are usually problem-free (at least in terms of threads) as there is only a single thread that can reference the process's resources. However, other programs will create multiple threads that then run in parallel in joint competition for the process's resources such as the CPU, and then also in competition with the threads of other processes. In such programs, it may be necessary to synchronize each thread's access to resources (such as a variable), and this is where thread problems come in.

In other words, a thread is the unit that the operating system schedules to run (meaning that the thread has a CPU to execute its instructions) and therefore a thread needs a data structure to store information when interrupted, a structure called TLS (*Thread Local Storage*), and each thread has its own stack. A thread only runs for a short time, after which it is interrupted by the operating system, allowing other threads to come into play. Since this time slice is only a few milliseconds, we experience it in the way that all programs (threads) are running in parallel. This interrupt can come at any time, and it is important to note that we as programmers do not know when it is happening and we may have to take special action in the program code to take into account that interrupts may occur. Also a program (thread) may be interrupted for reasons other than using the allocated time slice, but for example because the program performs an IO operation and has to wait until this operation is completed. The relationship between a process and threads can be illustrated as follows:



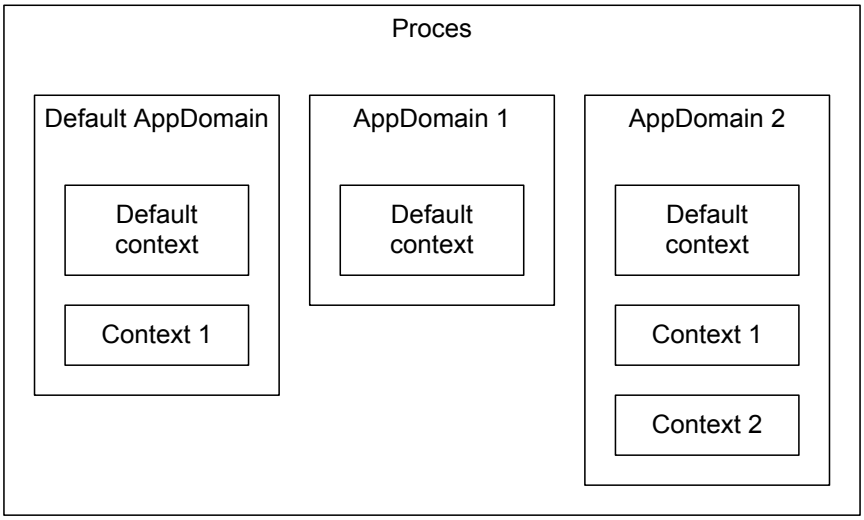
1.2 APPLICATION DOMAIN

As mentioned, a process can be interpreted as a running program where the program code is in an exe file or a dll. That is the code is in an assembly. The code needs to be loaded and is part of the program's process - or at least if it is unmanaged code and Win32 processes. Under .NET, it looks a bit different, since an assembly is hosted in a special partition in a process called a .NET *Application domain*. A process can easily contain multiple application domains, but for a given .NET program there will always be at least one. Application domains thus add a new concept to processes - a division of a process into isolated partitions.

There are three reasons for dividing a process into application domains:

1. Application domains are a .NET term that allows the .NET platform to be operating system neutral, so that the platform can be defined regardless of how a specific operating system handles processes and thus loads executable programs.
2. Application domain management is far simpler and requires far fewer resources than the administration of operating system processes. It is far faster for CLR to load and unload an application domain than to create a process, and this is of great importance for the runtime system's efficiency.
3. Application domains ensure even better application isolation than processes, and if an application within an application domain within a process fails, it will not affect other application domains within the same process.

One can therefore think of an application domain as a kind of .NET process and thus a kind of process that is exclusively managed by the runtime system. Typically, a simple .NET program will consist of a single application domain, called the default application domain. This will thus encapsulate the assemblies used by the program. An application domain is again divided into contexts. It is not a term that one see much as a programmer, but it is used by the runtime system to place different objects in different contexts, for example to deal with problems regarding concurrency. The relationship between a process, application domain and context can be illustrated as follows:



1.3 PROCESSES

The .NET Framework provides a number of types that represent processes and their properties, types found in the namespace *System.Diagnostics*. This section shows some examples that uses some of these types and shows what you can do with them. As example, I will use a class library with only one static class with two static methods:

```
namespace MathLIb
{
  public static class Functions
  {
    public static ulong Factorial(uint n)
    {
      if (n < 2) return 1;
      return n * Factorial(n - 1);
    }

    public static ulong Fibonacci(uint n)
    {
      if (n < 2) return n;
      ulong f1 = 0;
      ulong f2 = 1;
      while (n-- > 1)
      {
        ulong f3 = f1 + f2;
        f1 = f2;
        f2 = f3;
      }
      return f2;
    }
  }
}
```

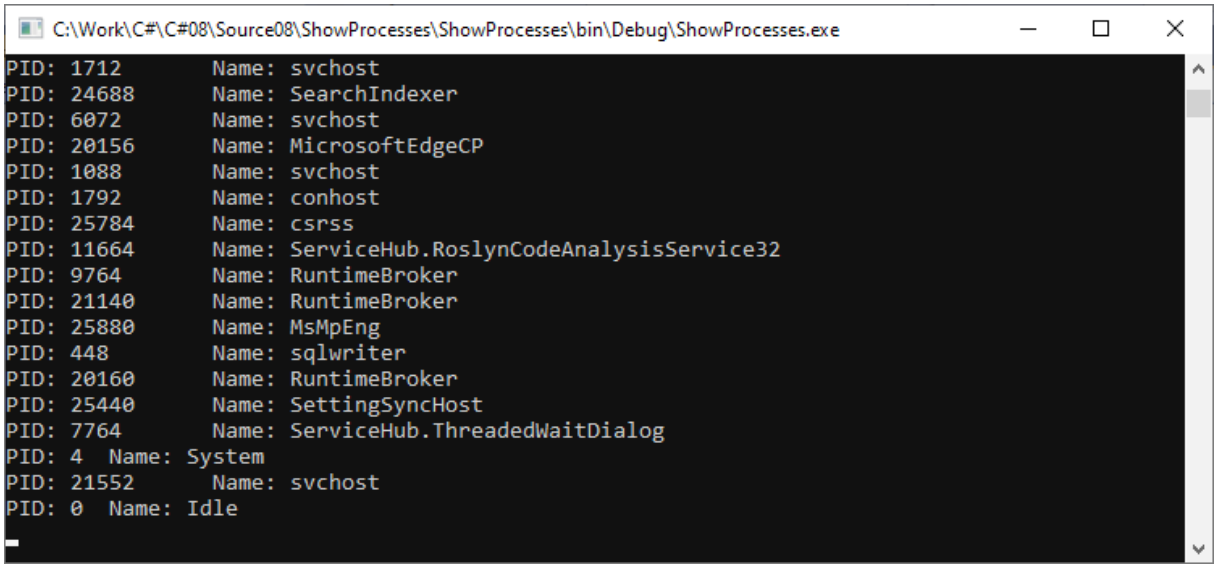
I also will use a program called *FibonacciNumbers*:

```
namespace FibonacciNumbers
{
  class Program
  {
    static void Main(string[] args)
    {
      for (uint n = 0; n <= 20; ++n)
      {
        Console.WriteLine(MathLib.Functions.Fibonacci(n));
        System.Threading.Thread.Sleep(1000);
      }
    }
  }
}
```

Note that the project needs a reference to the above class library. Also note that between each print is a *Sleep()* that is solely used to increase the run time. When the program runs it prints the Fibonacci numbers, one number each second.

I uses these types below, but first a program, which shows processes for running programs:

```
using System;
using System.Diagnostics;

namespace ShowProcesses
{
  class Program
  {
    static void Main(string[] args)
    {
      Process[] process = Process.GetProcesses(".");
      foreach (Process p in process)
        Console.WriteLine("PID: {0}\tName: {1}", p.Id, p.ProcessName);
      Console.ReadLine();
    }
  }
}
```



A process is represented by a class *System.Diagnostics.Process* that has a number of properties which provides information and the process. In this case, *Id*, which returns the process's PID is used as well as the *ProcessName* which returns the process name.

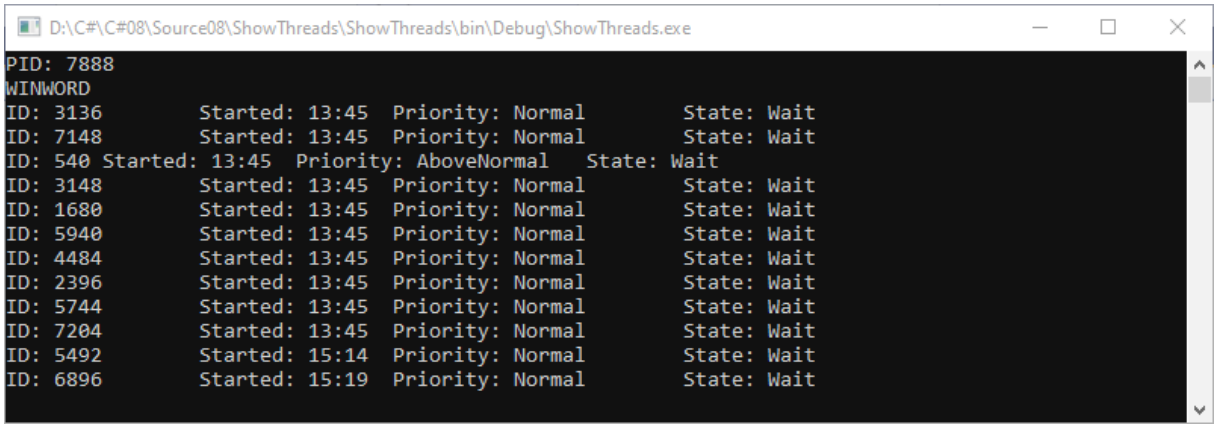
The class *Process* has a static method, which returns an array of *Process* objects. In this case, the parameter is ".", which means the local machine.

The next example (*ShowThreads*) shows the threads for a process and shows

- the threads id
- when the threads are started
- the thread priority
- the threads current state

```
static void Main()
{
  Console.Write("PID: ");
  int pid = Convert.ToInt32(Console.ReadLine());
  Process proc = Process.GetProcessById(pid);
  ProcessThreadCollection threads = proc.Threads;
  Console.WriteLine(proc.ProcessName);
  foreach (ProcessThread th in threads)
    Console.WriteLine("ID: {0}\tStarted: {1}\tPriority: {2}\tState: {3}",
      th.Id, th.StartTime.ToShortTimeString(), th.PriorityLevel,
      th.ThreadState);
}
```

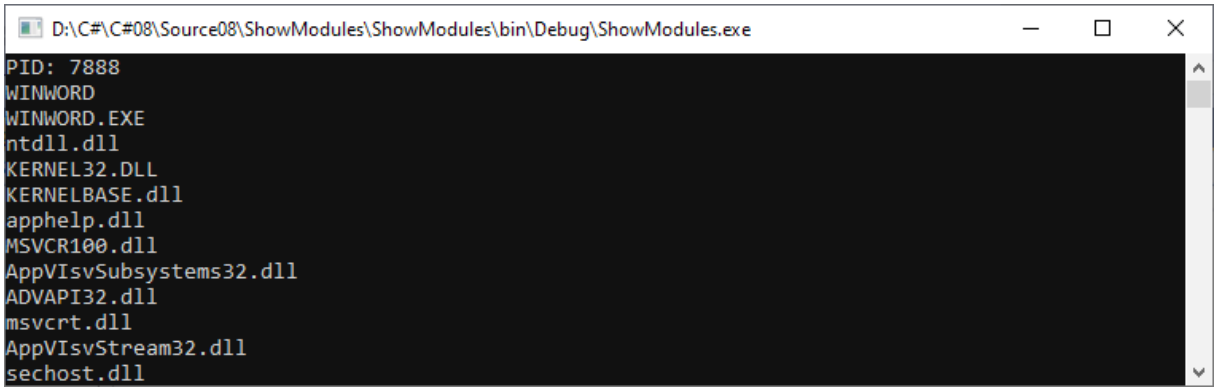
Note that to test the program, you must know the number of a process. Here you can either use *Task Manager* (Ctrl+Shift+Esc) in Windows, or you can use the above example. Below is an example showing which threads Microsoft Word has started:



The first part of the program consists of entering the process number, and then a static method in the *Process* class is used to obtain an object *proc* that represents that process. With this object available, you can get a collection of *ProcessThread* objects, each object representing a thread for that process. This collection is used to print information about the process threads.

In processes, a binary code module is called a module, and it can be both managed code (assemblies) and unmanaged code. Even simple programs will often have references to many modules. The following example (*ShowModules*) is in principle identical to the above, which prints an overview of a program's threads, but this time a summary of the program's modules is printed instead (again for Microsoft Word):

```
static void Main()
{
  Console.Write("PID: ");
  int pid = Convert.ToInt32(Console.ReadLine());
  Process proc = Process.GetProcessById(pid);
  ProcessModuleCollection modules = proc.Modules;
  Console.WriteLine(proc.ProcessName);
  foreach (ProcessModule md in modules) Console.WriteLine(md.ModuleName);
}
```



As another example the following starts two other programs:

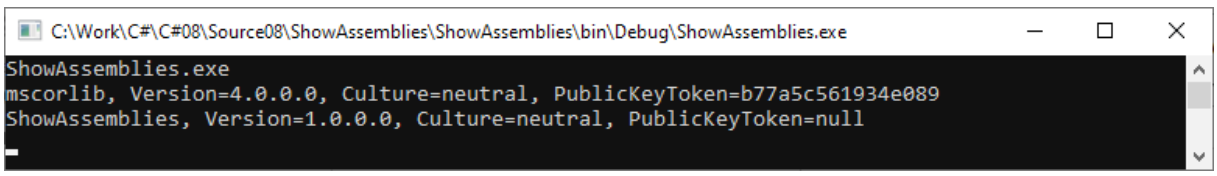
```
namespace StartProcess
{
  class Program
  {
    static void Main(string[] args)
    {
      Process proc1 = Process.Start("notepad.exe", "c:\\Temp\\Tekst.txt");
      Process proc2 = Process.Start("C:\\Temp\\FibonacciNumbers.exe");
      Console.WriteLine(proc1.Id);
      Console.WriteLine(proc2.Id);
      while (true)
      {
        Console.Write("PID: ");
        string pid = Console.ReadLine();
        if (pid.Length == 0) break;
        Process proc = Process.GetProcessById(int.Parse(pid));
        proc.Kill();
      }
      Console.ReadLine();
    }
  }
}
```

The class *Process* has a static method *Start()* used to start a program. The first parameter is the program name, and if there is a second parameter it is an argument for the started program. The first statement starts *Notepad* with the name of a text file as parameter while the other statement starts the above program *FibonacciNumbers*. If you you starts the program *StartProcess* the result is that three programs are started and *StartsProcess* prints the PID for two other programs:

You can the terminate the two programs by entering there PID as the class *Process* has a method *Kill()* to terminate a process.

The next example shows a program that prints the names of all assemblies that that program has loaded. Exactly, only those assemblies that are loaded into the process's default application domain, but in most cases it will also be the most important. The example should primarily show the use of the *AppDomain* class.

```
namespace ShowAssemblies
{
  class Program
  {
    static void Main(string[] args)
    {
      ShowAssemblier(AppDomain.CurrentDomain);
      Console.ReadLine();
    }

    static void ShowAssemblier(AppDomain domain)
    {
      Console.WriteLine(domain.FriendlyName);
      Assembly[] assem = domain.GetAssemblies();
      foreach (Assembly a in assem) Console.WriteLine(a.GetName().FullName);
    }
  }
}
```



The last example about processes show how to dynamically load and unload an assembly in the process application domain at runtime. Exactly, the program should load the two assemblies *MathLib.dll* and *FibonacciNumbers.exe*, but such that they are placed in each their application domain. The starting point is, as usual, a console application. Next, a reference

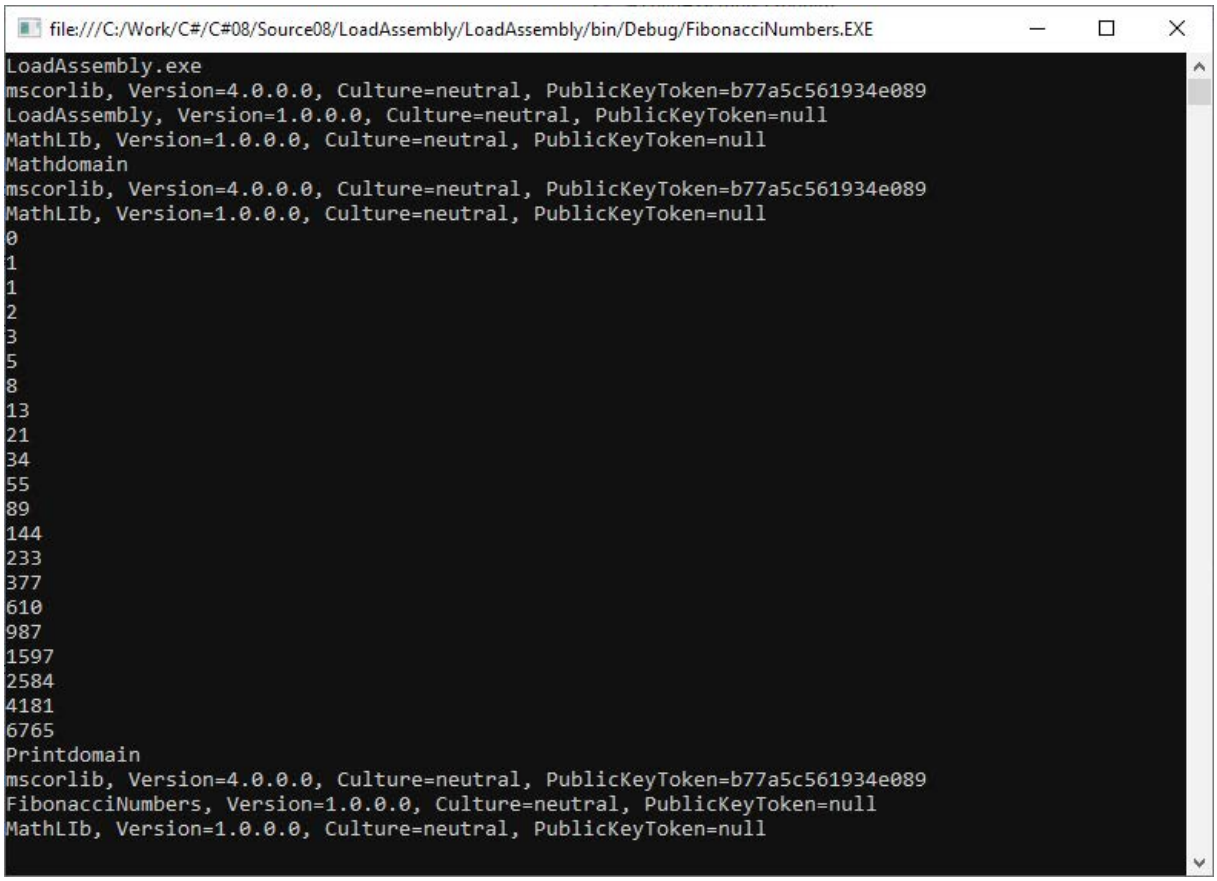
must be set to the two assemblies *MathLid.dll* and *FibonacciNumbers.exe*. Note that this means that the two assemblies are copied to the program's program library. The program can then be written as:

```
namespace LoadAssembly
{
  class Program
  {
    static void Main(string[] args)
    {
      AppDomain ad1 = AppDomain.CreateDomain("Mathdomain");
      ad1.Load("MathLib");
      ShowAssemblier(AppDomain.CurrentDomain);
      ShowAssemblier(ad1);
      AppDomain ad2 = AppDomain.CreateDomain("Printdomain");
      ad2.ExecuteAssembly("FibonacciNumbers.exe");
      ShowAssemblier(ad2);
      AppDomain.Unload(ad2);
    }

    static void ShowAssemblier(AppDomain domain)
    {
      Console.WriteLine(domain.FriendlyName);
      Assembly[] assem = domain.GetAssemblies();
      foreach (Assembly a in assem) Console.WriteLine(a.GetName().FullName);
    }
  }
}
```

First, a new application domain is created called *Mathdomain* and thus a form of a process. In this application domain the assembly *MathLib.dll* is loaded. This requires *MathLib* to be found in the program library, which is the case, since during the project a reference is added to the assembly. In this case, the assembly is not used for anything. This cannot be done immediately, as the program in the default application domain cannot directly communicate with or refer to an assembly in another application domain, but there are techniques that make it possible.

Next, the program prints which assemblies is loaded in the two application domains, and next, another application domain is created and the process now has three application domains. In the last domain, the program *FibonacciNumbers.exe* is loaded and the program is started by running the *Main()* method. The result is that the process now runs two applications, but they each run in their own application domain.

You cannot directly unload an assembly, but you can unload a secondary application domain and thus also the assemblies that that domain has loaded



2 THREADS

For programs that only start a single thread, everything about threads are uninteresting, but for multi-thread programs it is different and threads are something that can make life difficult for a programmer. A thread represents the unit that the operating system schedules (chooses to run on the processor), and a thread can therefore be interrupted by the operating system. This means that you have no control over when a thread is running, which is why synchronization mechanisms may be needed to control that threads do not interfere with each other's work. That's all what, that can make working with threads difficult.

There are several reasons why you are interested in running a program as multiple threads, but basically it is about parallelism, where for the user, when a program's threads are running in turn, it looks like the program is performing operations in parallel. Although most machines today are multiprocessor machines (or at least have multiple cores) and thus a machine that can actually run multiple threads at the same time, in practice there will always be pseudo-parallelism where one thread has the CPU for a very short period of time to be interrupted and placed in a wait queue until it again gets a time on the CPU. The reason for this pseudo-parallelism is that the number of threads will always far exceed the number of processors. A typical reason for allowing a program to perform a particular operation in its own thread may be a program that needs to perform demanding data processing (for example a spell check, input data from an external device, etc.). If the operation is performed in its own thread, the user can continue to use the program while that operation is being performed. An example might be a program that runs an animation, an animation that runs while the user is working with the program. This is only possible if the animation is running in its own thread.

There is no direct relation between threads and application domains, but at some point an application domain may have multiple threads running. An active thread cannot immediately do anything in another application domain (it can only work in one application domain at a time), but it can be moved to another application domain by CLR. It is rare that it is something you are interested in as a programmer, but the opportunity is there.

The types for threads belong in the namespace *System.Threading*, but instead of starting directly with threads I want to remind you of asynchronous delegates from the book C# 4, which is both an easy and a relatively safe way to implement multi-thread programs.

2.1 ASYNCHRONOUS DELEGATES AGAIN

A delegate is a type that defines variables that can refer to a method - and possibly several methods. The method referred to by a delegate can be executed via the delegate, and the method can be performed synchronously or asynchronously by the delegate, the latter means that the method runs in its own thread. This section will give a few examples of how to start threads using an asynchronous delegate.

Sorting an array is a classic problem in programming, and a number of methods have been worked out to solve the problem, methods that differ in efficiency, some being much faster than others, and the difference becomes more pronounced the larger the array is. Three well-known methods go under the following names

- bubble sort
- selection sort
- insertion sort

The three methods are similar in that they are all very simple and are quite similar in terms of efficiency and are not very effective - at least not for large arrays.

The following example creates three arrays of the same size. Next, each of the three arrays is sorted, but with three different sorting methods, and the program prints for each method how long it has taken. Each sort is started in a separate thread, so that the three sorts are executed in parallel next to each other and again in parallel with the main thread running the *Main()* method.

It is not the sorting methods them self and how they work, that is interesting (and the code is not shown below), but the important thing is that it is methods that take some time, so you find that these are methods that run parallel side by side. The program is written in the following way:

```
namespace Sorting
{
  class Program
  {
    const int N = 20000;
    static Random rand = new Random();

    delegate void Sorting(int[] v);
    delegate string WorkToDo(int[] v, Sorting sort, string name);

    static void Main(string[] args)
    {
      Console.WriteLine("Primary thread: " + Thread.CurrentThread.
      ManagedThreadId);
      WorkToDo work1 = new WorkToDo(Sort);
      WorkToDo work2 = new WorkToDo(Sort);
      WorkToDo work3 = new WorkToDo(Sort);
      IAsyncResult res1 = work1.BeginInvoke(Create(N, 0, 2 * N),
        SelectionSort, "Selection sort", null, null);
      IAsyncResult res2 = work2.BeginInvoke(Create(N, 0, 2 * N),
        BubbleSort, "Bubble sort", null, null);
      IAsyncResult res3 = work3.BeginInvoke(Create(N, 0, 2 * N),
        InsertionSort, "Insertion sort", null, null);
      while (!res1.IsCompleted || !res2.IsCompleted || !res3.IsCompleted)
      {
        Console.WriteLine("Main() running!");
        for (uint u = 0; u < 1000000; ++u) Math.Sin(Math.Log((Math.
        Sqrt(2))));
      }
      Console.WriteLine(work1.EndInvoke(res1));
      Console.WriteLine(work2.EndInvoke(res2));
      Console.WriteLine(work3.EndInvoke(res3));
      Console.ReadLine();
    }

    static string Sort(int[] v, Sorting sort, string name)
    {
      int id = Thread.CurrentThread.ManagedThreadId;
      Console.WriteLine("{0} started as thread {1}", name, id);
      Stopwatch sw = new Stopwatch();
      sw.Start();
```

```
      sort(v);
      sw.Stop();
      return string.Format("{0} terminated in thread {1} after {2}
      milliseconds",
        name, id, (int)sw.Elapsed.TotalMilliseconds);
    }

    static void SelectionSort(int[] v) { ... }

    static void BubbleSort(int[] v) { ... }

    static void InsertionSort(int[] v) { ... }

    static int[] Create(int n, int a, int b)
    {
      int[] v = new int[n];
      for (int i = 0; i < n; ++i) v[i] = rand.Next(a, b);
      return v;
    }
  }
}
```

Some of the code does not have much to do with threads, but concerns the sorting methods and is not shown above. The last method is an auxiliary method which creates an *int* array with n integers in the interval from *a* to *b*. The method does not contain anything special and is used to create the arrays to be sorted. A sorting algorithm will always consist of traversing the array (several or many times), and then compare and possibly swap elements that are arranged incorrect. The goal is to have as few passes as possible and avoid switching more elements than necessary. Above are shown three sorting methods that work that way, and the difference is solely how the array is traversed. I do not want to go into a closer analysis of the three algorithms here, but I can mention that for practical purposes they have the same efficiency and that they are all ineffective for large arrays.

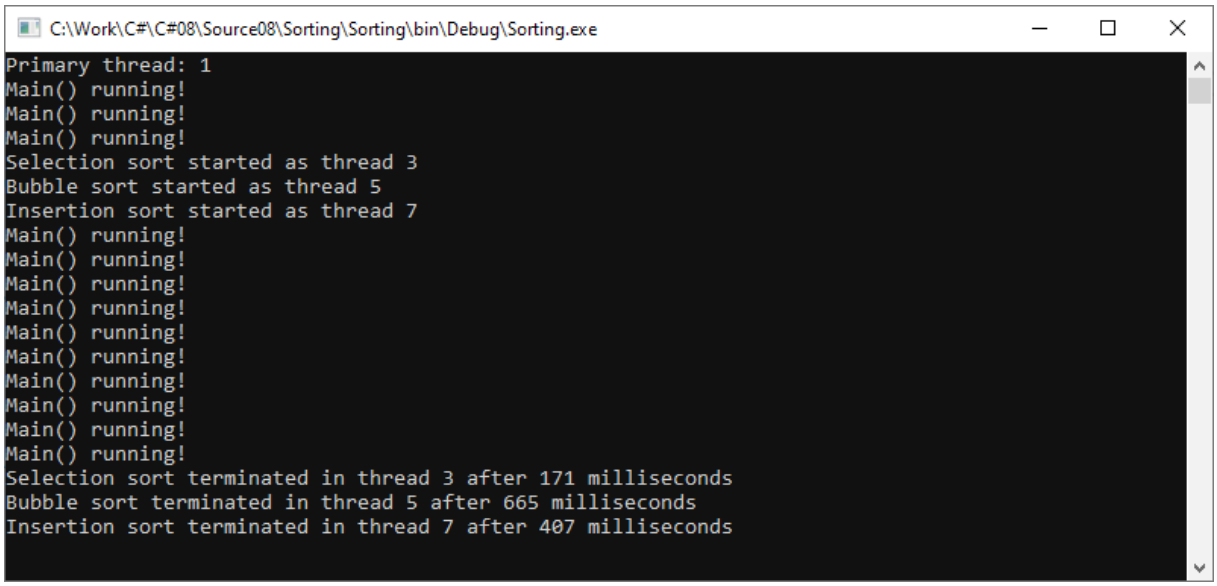
The method *Sort()* has three parameters, the first one is the array to be sorted, the next one is a delegate used to point to the sorting method to be used, and finally the last parameter is a text that the method prints and applies to the return value.

The class *Thread* has a static property called *CurrentThread* which represents the current thread. It is used here to refer to the thread ID, and the *Sort()* method starts by printing the value of the parameter *name* and the thread ID. Next, the time is measured with a *StopWatch* object, after which the sorting method is performed. When it terminates, the stopwatch is stopped and the method returns a string that tells how long the operation took.

Finally, there is the *Main()* method, like the place where it all happens. *WorkToDo* is a delegate with the same signature as the method *Sort()*. First, three references to the *Sort()* method of type *WorkToDo* are created. Next, the methods that the three delegates point to are executed, but so that they are started as an asynchronous delegate and thus run in their own threads. Each thread will perform its own sorting, but the threads each use their own sorting method. The result is that the program now runs four threads, with the three being the sorting methods, the last being the primary thread. It runs in a loop until the three sorting methods are completed. For each iteration, the loop prints a text and does some "dummy" work - just to do some work that takes time. So you should note that the primary thread is not just suspended, but that it is constantly using the processor.

When the sorting is completed, the program prints the result (time for) the three sorting operations. So you should note that there are four threads running and that they are threads that constantly execute instructions and therefore need a processor and that all the threads take time. This means that the threads alternate, but we feel that the threads run parallel to each other.

If you run the program, the result could be (on my machine and where N = 10000):



You must note that in this example the three threads do not use shared resources, each thread sort its own array. Therefore, there is no need for synchronization of the three threads.

EXERCISE 1: FIND PRIMES

Write a console application which you can call *FindPrimes*. Add a static method

```
static long Next()
{
  return (((long)rand.Next(int.MaxValue)) << 32) + rand.Next(int.
MaxValue);
}
```

That returns a random positive *long*. Add a method

```
static void Find(long n)
{
}
```

The method must print the current thread id, the number *n* and the first prime which are greater than or equal *n*. The idea is that it take some time to execute the method for big values of *n*.

Write a method

```
static void Test1(List<long> list)
{
}
```

when the method must start a stopwatch (as a *Stopwatch* object). The method must then call the method *Find()* for each numbers in the list (the parameter), and after these operations stop the stopwatch and print the number of milliseconds used to perform all the *Find()* operations.

Write the following *Main()* method:

```
static void Main(string[] args)
{
  List<long> list = new List<long>();
  for (int i = 0; i < N; ++i) list.Add(Next());
  Test1(list);
  Console.ReadLine();
}
```

where *N* is a variable which indicates the number of items in *list* (for example 10).

You must then write another method:

```
static void Test2(List<long> list)
{
}
```

which must du the same as *Test1()*, but with the difference that the method *Find()* must be executed as an asynchronous delegate. There is a problem to determine when all the methods are finished, and here you can in a busy wait with the methods' *AsyncResult* test whether they are all completed before you stop the stopwatch. It is not the right way to solve the problem, but accept it in this case.

In *Main()* you must also call *Test2()*. Can you observe any difference in time to perform the two methods and if so what is the reason for the time difference? Also try experimenting with the value of *N*, both values less than 10 and values greater than 10.

2.2 THE CLASS THREAD

Above I have shown how to start a thread with an asynchronous delegate, which is quite easy, and it is often a good way, among other things because it is simple to pass parameters to the thread (the method *Sort()* above). However, you can also start a thread directly using the class *Thread*, which allows you to gain better control over how the thread runs, but conversely, working with threads at a lower level is also an option.

A thread is always started by a method (static or non-static). It must be either a void method without parameters or a void method with a parameter of the type *object*. The thread will run until the method terminates, or until the thread is terminated otherwise.

A thread is started as a foreground thread, which means that the primary thread only stops (the program terminates) when all secondary threads are terminated, which is probably the behavior you are typically interested in. However, there are other options, as a thread can also be started as a background thread, and such threads are terminated by the runtime system when the primary thread terminates, and they are immediately killed regardless of what they are doing. That is the primary thread does not wait for background threads to terminate - unless explicitly requested.

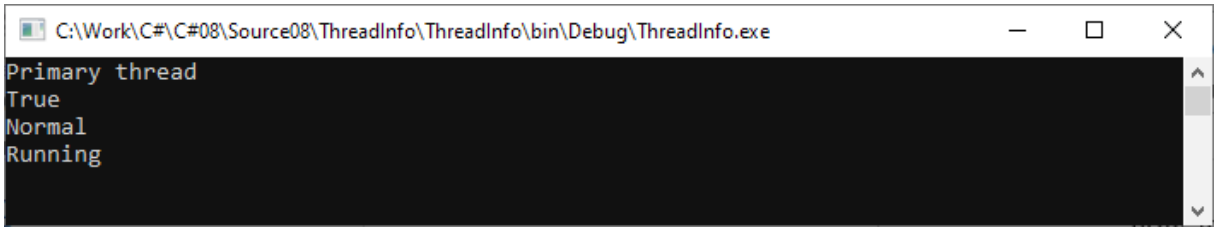
If you do nothing, threads run without any synchronization and you have no control over when a thread runs. The thread is interrupted when suspending itself, but it can also be interrupted at other times by the operating system, and it is these conditions that make programming multi-thread programs cumbersome and that may require special actions to synchronize threads processing of for example variables or other shared resources.

In this section, I will primarily look at how to start threads using the class *Thread*. In addition to showing how to create a thread as a *Thread* object, the examples should shows how to pass parameters to the thread method, how to block the primary thread until the secondary threads end, and the difference between a foreground thread and a background thread.

The first example prints information about the primary thread:

- the thread's name
- where the thread is alive
- the thread's priority
- the thread's state

```
namespace ThreadInfo
{
  class Program
  {
    static void Main(string[] args)
    {
      Thread th = Thread.CurrentThread;
      th.Name = "Primary thread";
      Console.WriteLine(th.Name);
      Console.WriteLine(th.IsAlive);
      Console.WriteLine(th.Priority);
      Console.WriteLine(th.ThreadState);
    }
  }
}
```

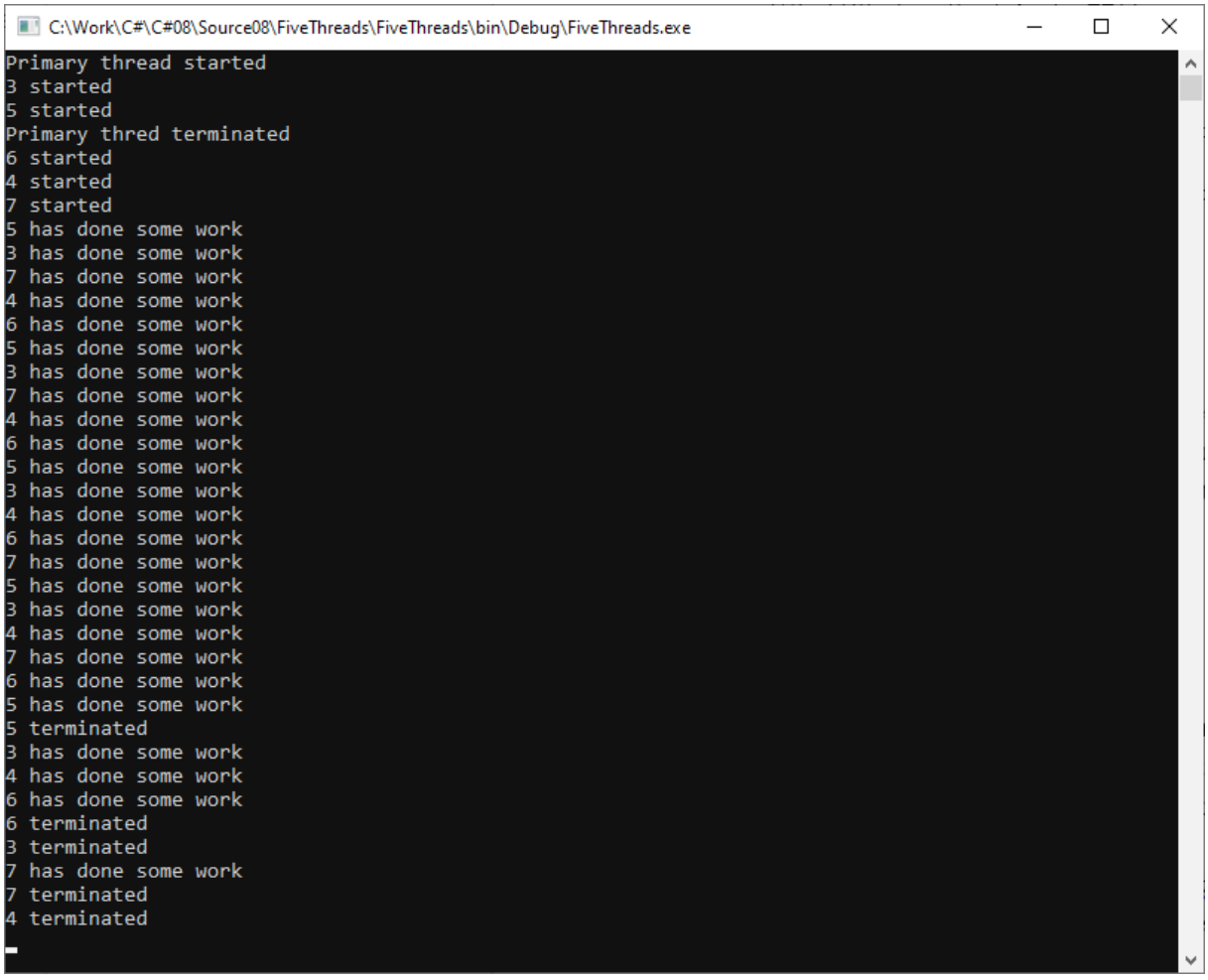


You reference a thread's priority with the property *Priority*. A thread is usually started with the *Normal* priority, but there are the following options:

- Lowest
- BelowNormal
- Normal
- AboveNormel
- Highest

The next example (*FiveThreads*) starts five threads in addition to the primary thread:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Primary thread started");
    Thread[] th = new Thread[5];
    for (int i = 0; i < th.Length; ++i) th[i] =
      new Thread(new ThreadStart(WorkToDo));
    for (int i = 0; i < th.Length; ++i) th[i].Start();
    Console.WriteLine("Primary thread terminated");
  }

  private static void WorkToDo()
  {
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId + " started");
    for (int i = 0; i < 5; ++i)
    {
      for (uint u = 0; u < 1000000; ++u) Math.Sin(Math.Log((Math.Sqrt(2))));
      Console.WriteLine(
        Thread.CurrentThread.ManagedThreadId + " has done some work");
    }
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId + "
    terminated");
  }
}
```



The example shows how to start a thread, and partly that several threads run in parallel that compete for the processor. You should note that the program does not stop until the five threads are completed, but also that the last statement in the *Main()* method is executed before the threads terminate, that is *Main()* does not wait for its threads to finish.

The class *Thread* represents a thread and you create a thread as an object of the type *Thread*. The class's constructor has one parameter that has the type *ThreadStart*, which is actually a delegate that refers to the method to be executed by the thread. Once the thread is created, it is started with the method *Start()*.

The individual threads each perform their own *WorkToDo()*. It is a method that runs a short loop where each repetition consists of a calculation (which takes time) as well as the printing of a message on the screen.

You should note that it is simple to create a thread as an instance of the *Thread* class using the delegate *ThreadStart*. This delegate refers to the method (usually called the thread method) that needs to be started as a thread, but the delegate does not allow to pass parameters to the thread method. How to do that is shown in a later example.

If you want *Main()* to wait for the threads you can do that with *Join()* (the example *JoinThreads*):

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Primary thread started");
    Thread[] th = new Thread[5];
    for (int i = 0; i < th.Length; ++i)
      th[i] = new Thread(new ThreadStart(WorkToDo));
    for (int i = 0; i < th.Length; ++i) th[i].Start();
    Console.WriteLine("Primary thread terminated");
  }

  private static void WorkToDo()
  {
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId + " started");
    for (int i = 0; i < 5; ++i)
    {
      for (uint u = 0; u < 1000000; ++u) Math.Sin(Math.Log((Math.Sqrt(2))));
      Console.WriteLine(
        Thread.CurrentThread.ManagedThreadId + " has done some work");
    }
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId + "
    terminated");
  }
}
```

Compared to the previous example, there is only one difference:

```
for (int i = 0; i < th.Length; ++i) th[i].Join();
```

For each thread a *Join()* is executed. This means that the current thread, which here is the primary thread, is blocked until the thread that has executed a *Join()* terminates. When this happens for all five secondary threads, the result is that the primary thread waits until the 5 secondary threads finish, and the final *Console.WriteLine()* is performed only after the 5 threads are completed.

Above I have created threads with the delegate *ThreadStar*, but there is an alternative called *ParameterizedThreadStart* that allows you to pass parameters to the thread method. In the following example (*ThreadParameters*), I will write a method called with an interval as parameter, and then the method must print all prime numbers within that range. The method must run in its own thread, and the challenge then is how I get the interval passed as a parameter to the method.

This is where *ParameterizedThreadStart* enters the image as a delegate to start a thread where a parameter is passed. However, you can only transfer one parameter which always has the type *object*, and if there are several, as is the case here, it is necessary to encapsulate the parameters in a data structure. I will therefore start with the following parameter type:

```
struct Param
{
  public ulong a, b;

  public Param(ulong x, ulong y)
  {
    a = x;
    b = y;
  }
}
```

which represents a range from a to b. With this type in place, you can write the thread method:

```
private static void Primes(object param)
{
  Param p = (Param)param;
  int id = Thread.CurrentThread.ManagedThreadId;
  for (ulong n = Prime(p.a); n <= p.b; n = Prime(n + 2))
    Console.WriteLine("Thread {0}: {1}", id, n);
}
```

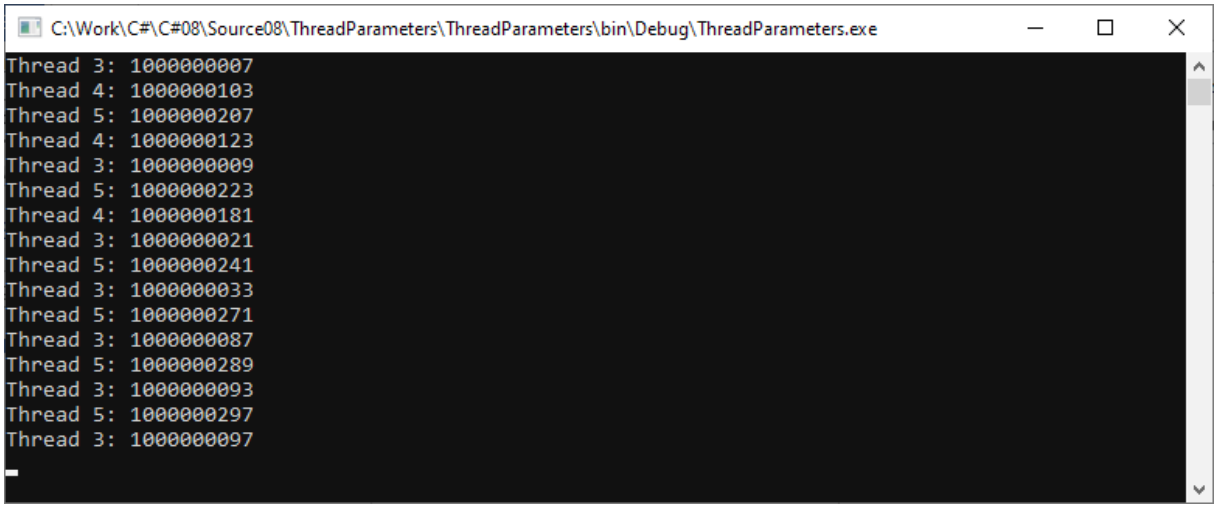
The method prints all prime numbers within an interval defined by the parameter *param*. Since the type of this parameter is *object*, a type of cast is required for a *Param*. The method uses an auxiliary method *Prime()*, which returns the first prime number that is greater than or equal to the parameter *n*. The important thing here is that if *n* is large, the method takes a long time.

```
public static ulong Prime(ulong n)
{
  if (n <= 2) return 2;
  ulong k = 3;
  while (k < n || !IsPrime(k)) k += 2;
  return k;
}

public static bool IsPrime(ulong n)
{
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (ulong k = 3, m = (ulong)Math.Sqrt(n) + 1; k <= m; k += 2)
    if (n % k == 0) return false;
  return true;
}
```

Back there is the *Main()* method, which creates three threads that each print the prime numbers within an interval. The threads are created with the parameter *ParameterizedThreadStart* with reference to the thread method *Primes()*. Next, the threads are started, but this time with a parameter that specifies the interval:

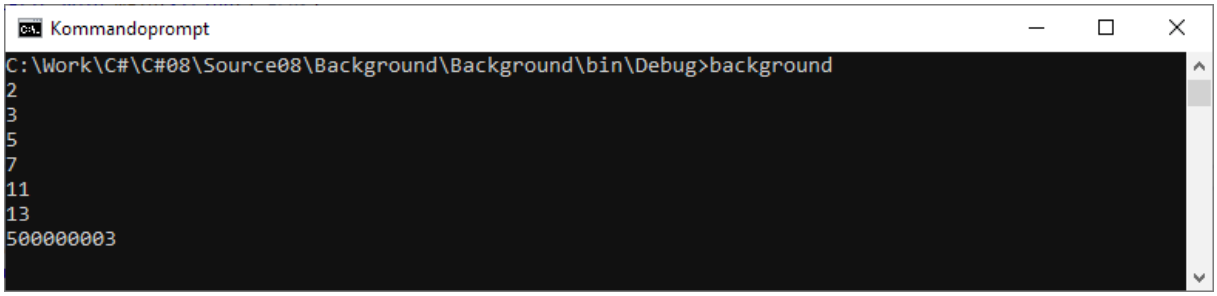
```
static void Main(string[] args)
{
  Thread t1 = new Thread(new ParameterizedThreadStart(Primes));
  Thread t2 = new Thread(new ParameterizedThreadStart(Primes));
  Thread t3 = new Thread(new ParameterizedThreadStart(Primes));
  t1.Start(new Param(1000000001, 1000000100));
  t2.Start(new Param(1000000101, 1000000200));
  t3.Start(new Param(1000000201, 1000000300));
}
```

Running the program results in the following:



As the last example on how to start and running threads I will show how to start a *backlground thread*. All the above examples has been *foreground threads*. The following program (called *Background*) starts two threads, one as a background thread, and the other using a delegate that always starts as a foreground thread:

```
class Program
{
  private delegate ulong Calculator(ulong n);

  static void Main(string[] args)
  {
    Thread th = new Thread(new ThreadStart(Print));
    th.IsBackground = true;
    th.Start();
    Calculator calc = new Calculator(Prime);
    IAsyncResult res = calc.BeginInvoke(500000000, null, null);
    Console.WriteLine(calc.EndInvoke(res));
  }

  private static void Print()
  {
    for (ulong n = Prime(1); true; n = Prime(n + 1))
    {
      Console.WriteLine(n);
      Thread.Sleep(100);
    }
  }
}
```



First, a delegate is defined used to execute a method in its own thread with an asynchronous delegate. The method is *Prime()* from the previous example.

In *Main()*, a thread *th* is created for the method *Print()*. It is a method that prints all prime numbers, but in order not to run too fast, a delay of 1/10 second is inserted after each number. After the thread is created, it is defined as a background thread (a thread is by default a foreground thread):

```
th.IsBackground = true;
```

After starting the thread, a delegate is created with reference to the method *Prime()*, and it is started as an asynchronous delegate, and must determine the first prime number greater than 500000000, which takes a few seconds.

The result is that the method *Print()* starts running in a background thread, and at the same time *Prime()* starts with the delegate. When *Prime()* is completed, the return value is sent to *Main()* where it is printed, and then the program terminates regardless of whether *Print()* is still running - the primary thread does not wait for background threads to be completed.

EXERCISE 2: SORTING

In this exercise you should write a program that in principle does the same as the example *Sorting* from the previous section.

Create a new console application project which you can call *Sorting*. Copy the 5 methods:

1. *SelectionSort()*
2. *BubbleSort()*

3. *InsertionSort()*
4. *Swap()*
5. *Create()*

from the previous *Sorting* project. Modify the three sorting methods (the first three methods) such they all as the last statement print a message, for example:

```
Console.WriteLine("Selection sort end");
```

Add a method

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

that prints the elements in the array *v* in one line on the screen. Also add a method that creates and return a copy of an array:

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

To test the three sorting methods you must create a method *Test1()* which

- creates an *int* array with 20 2-digit numbers
- create two copies of this array
- print the first array
- sort the first array with *SelectionSort()*
- print the array again
- do the same for the two other arrays but instead using *BubleSort()* and *InsertionSort()*

Perform the method from *Main()*. Note that the only reason for the method *Test1()* is to validate that the three sorting methods work properly.

You must then write a method

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

which in the same way creates three arrays and sort them, where *n* is the number of elements in the arrays. The method should not print the arrays, but instead it with a stopwatch must measure the number of milliseconds it takes to sort the three arrays using the three sorting methods.

As the next step you must write a method *Test3()* which in the same way as *Test2()* creates an array and two copies and sorts the arrays, but this time the arrays must be sorted in each there thread. To start one of the sorting method in its own thread you have to transfer the array to sort as a parameter. You must then change the three sorting methods such there parameter is an object, for example:

```
static void SelectionSort(object obj)
{
  int[] v = (int[])obj;
```

You can the start the sorting methods as a thread using a *ParameterizedThreadStart*. To ensure that you do not read the stopwatch until the three threads are finished, it is necessary to join the primary thread to the three threads.

Add a variable

```
static int count = 3;
```

Update each of the three sorting methods such they as the last statement performs the following statement:

```
--count;
```

As the last example add the following method:

```
static void Test4(int n)
{
  Stopwatch sw = new Stopwatch();
  int[] v1 = Create(n, n, 2 * n);
  int[] v2 = Copy(v1);
  int[] v3 = Copy(v1);
  Thread th1 = new Thread(new ParameterizedThreadStart(SelectionSort));
  Thread th2 = new Thread(new ParameterizedThreadStart(BubbleSort));
  Thread th3 = new Thread(new ParameterizedThreadStart(InsertionSort));
  th1.IsBackground = true;
  th2.IsBackground = true;
  th3.IsBackground = true;
  sw.Start();
  th1.Start(v1);
  th2.Start(v2);
  th3.Start(v3);
  for (int i = 0; i < 2; ++i)
  {
    Thread.Sleep(100);
    Console.WriteLine("Waiting");
  }
  if (count == 0)
  {
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
  }
}
```

The method performs the same as *Test3()*, but this time the threads are started as background threads. The method should shove that if the *for* loop terminates before the sorting threads are finish then the program stops without waiting for the three threads. To test this behavior you must remove a possible

```
Console.ReadLine();
```

in *Main()* and the start the program from a prompt.

You should note that the program has a concurrency problem as three threads use the same variable *count*. I do not think that you will observe a problem in this case, but it is the subject for the next cpapter.

EXERCISE 3: STARTING THREADS

In this exercise you must write a program which starts 5 threads. The purpose is to practice the syntax to start a thread. Create a new console application project as you can call *StartingThreads*.

Write a method

```
static void Fibonacci()
{
}
```

which in a loop prints the first 50 Fibonacci numbers:

```
1 2 3 5 8 13 ... 20365011074
```

when each iteration of the loop must perform a sleep in a random time between ½ and 1 second. In *Main()* you must perform the method in a *Thread*:

```
(new Thread(new ThreadStart(Fibonacci))).Start();
```

Start another thread which executes the same method but this time without indicate a *ThreadStart* delegate, but instead using a lambda expression.

Write a method

```
static void Sum(object obj)
{
}
```

where the parameter should be interpreted as an array of the type *double*. The method must in a loop calculate the sum of the elements in the array and as the last statement print the sum. For each iteration of the loop the method must perform a sleep in a random time between 1 and 2 seconds. In *Main()* you must start a thread which executes the method:

```
(new Thread(new ParameterizedThreadStart(Sum))).Start(
   new double[] { Math.Sqrt(2), Math.Sqrt(3), Math.Sqrt(5) });
```

Start another thread which executes the same method with the same parameters, but this time without indicate a *ParametererizedThreadStart* delegate, but instead using a lambda expression.

Sometimes you need a thread that returns a value. One way to solve this problem is to define a variable where the thread can store the value, but a simpler way is to use an asynchronous delegate. Write a method which calculates and return the product of the elements in an array:

```
static double Product(double[] v)
```

Executes this method from *Main()* as an asynchronous delegate and print the result.

3 CONCURRENCY

As shown above, starting a thread is relatively simple, and at least a program can do it in three ways:

1. using an asynchronous delegate
2. with the *Thread* class and the *ThreadStart* delegate
3. with the *Thread* class and the *ParameterizedThreadStart* delegate

Threads are used to create parallelism in a process where several activities are performed simultaneously. Actually, this is pseudo-parallelism, since the CPU can only execute one thread at a time, and the parallelism occurs because the CPU switches between the threads very quickly. One can think of it in the way that the individual threads alternately uses the CPU for a very short period of time, after which they are interrupted and a new thread arrives. The idea is that this shift happens so quickly that we as users experience it as if a thread is running all the time - threads are constantly making progress. Today, most machines have more CPUs or at least more cores, but that does not change the image as there will always be far more threads than CPUs.

This shift between when a thread is running and blocked poses challenges to a multi-threaded program, as one cannot know when a thread will be interrupted. In fact, it can happen in the middle of an assignment statement. All threads within the same process (application domain in .NET) have access to all the data of the process, and you can for example have multiple threads that modify the same variable, and therefore one thread cannot be sure that the value of a variable has not been changed by another thread. Therefore, it may be necessary to synchronize thread's access to common resources, which actually means that a given sequence of instructions is always executed as a whole, or more that only the thread's instructions can use that resource. To that end, there are several synchronization mechanisms that can be used by the programmer, and they differ primarily in terms of ease of use, but also in terms of flexibility and efficiency. In the following, I will show the use of the following mechanisms:

- Lock
- Monitor
- Interlock
- Mutex
- Semaphor

and they all have the purpose of locking one resource so that only one thread can use the resource. It is not very difficult, but if you can set locks, you must also be able to unlock again, and that is exactly where the problem is. You can lock too much, and then it will go beyond performance, and you can lock more resources in such a way that no thread can do anything, and then the program is run in a deadlock situation and hangs.

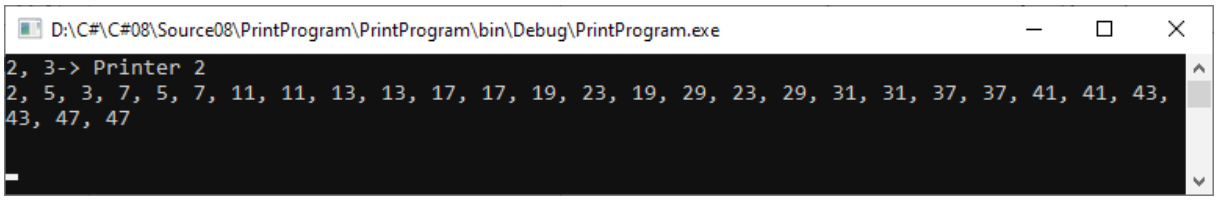
The following example should illustrate what it means that threads are unsynchronized. The program is called *PrintProgram* and it has a class with a method *Print()* that prints all prime numbers less than 50. The prime numbers themselves are laid out in a static array:

```
class Printer
{
  private static Random rand = new Random();
  private static int[] t =
    { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 };

  public virtual void Print()
  {
    Thread.Sleep(rand.Next(1000));
    Console.WriteLine("-> {0}", Thread.CurrentThread.Name);
    Console.Write(t[0]);
    for (int i = 1; i < t.Length; i++)
    {
      Thread.Sleep(rand.Next(2000));
      Console.Write(", " + t[i]);
    }
    Thread.Sleep(rand.Next(1000));
    Console.WriteLine();
  }
}
```

The method *Print()* prints the name of the thread which executes the method. Then the prime numbers are printed, but before a number is printed the thread is suspended in max 2 seconds. The *Main()* method instantiate a *Printer* object and creates two secondary threads performing the method *Print()*:

```
class Program
{
  static void Main(string[] args)
  {
    Printer p = new Printer();
    Thread th1 = new Thread(new ThreadStart(p.Print));
    Thread th2 = new Thread(new ThreadStart(p.Print));
    th1.Name = "Printer 1";
    th2.Name = "Printer 2";
    th1.Start();
    th2.Start();
    Console.ReadLine();
  }
}
```

If you execute the program, the result could be as shown below. It should be noted that since the two threads are unsynchronized, they will print alternately so that the result is mixed together.



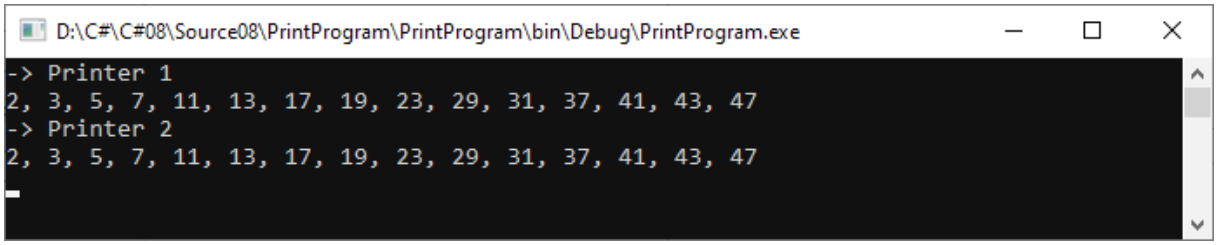
To synchronize the two methods I can use a *lock*. It is a reserved word in C# and is used to put a lock on a code block. The result is that there is only one thread that can execute that block. If a thread is inside the block - it is said that the thread has the lock - and if another thread tries to execute the block at the same time - it is said that it is trying to get the lock - then the second thread is blocked and has to wait until the first thread is finished and releases the lock. A thread releases the lock when it exits the locked code block.

To show how it works I have written a class *Printer1*. The class inherits the class *Printer*, but it implements a thread safe version of the method *Print()*:

```
class Printer1 : Printer
{
  private object lockObject = new object();

  public override void Print()
  {
    lock (lockObject)
    {
      base.Print();
    }
  }
}
```

Locking requires an object that one can lock. It does not matter what the object is, and usually you use an object of the type *object*. You can then lock a block as shown in the method *Print()* above. That is the call that the base class's *Print()* method is locked, and there is thus only one thread that can use the base class's *Print()* method.

In *Main()* I have instantiated a *Printer1* object instead of a *Printer* object, and if I run the program the result is:



3.1 A MONITOR

In this section I will show an example that is basically the same as the above, but it should show that even if the *Print()* method is executed as a whole, the threads that execute the method may well be interrupted. It simply means that the second secondary thread (the one trying to get the lock) of the operating system is put in a queue until the first thread (the one having the lock) is finished. Here it is illustrated by starting a third secondary thread that changes the background color of the console every second. You can thus see that the third thread is running at the same time as the *Print()* methods are running.

However, the *Printer1* class has been changed so that a monitor is used instead of a lock. The class still inherits the class *Printer*, but instead of a *lock* it locks with a *monitor*. A lock is really just a monitor, and a monitor works in the same way as a lock. That is it sets a lock so that only one thread can execute the code until the monitor releases the lock. The advantage of a monitor over a lock is that the monitor provides some extra options. For example a monitor can explicitly using a *Pulse()* or *PulseAll()* send notifications to pending threads, and it can suspend itself with a *Wait()*. In return, it is the programmer's responsibility to release the lock.

```
class Printer1 : Printer
{
  private object lockObject = new object();

  public override void Print()
  {
    Monitor.Enter(lockObject);
    try
    {
      base.Print();
    }
    finally
    {
      Monitor.Exit(lockObject);
    }
  }
}
```

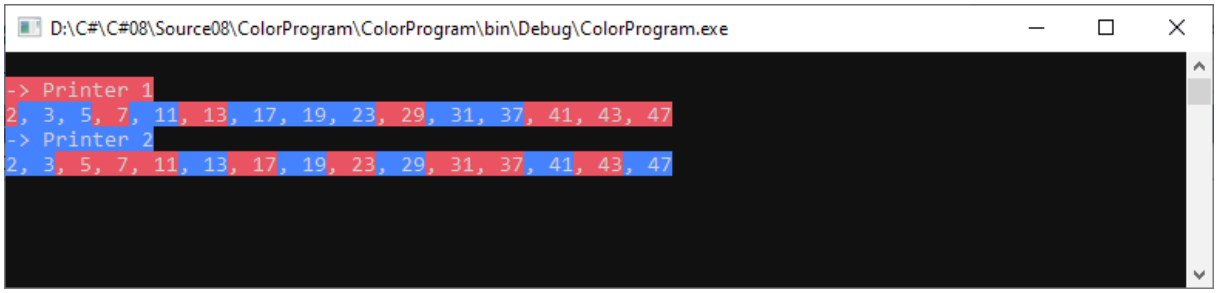
To make sure that the monitor releases the lock with *Exit()*, this statement is placed in a finally block.

The main class is:

```
class Program
{
  static void Main(string[] args)
  {
    Console.ReadLine();
    Printer p = new Printer1();
    Thread th1 = new Thread(new ThreadStart(p.Print));
    Thread th2 = new Thread(new ThreadStart(p.Print));
    Thread th3 = new Thread(() => ChangeColor());
    th3.IsBackground = true;
    th1.Name = "Printer 1";
    th2.Name = "Printer 2";
    th1.Start();
    th2.Start();
    th3.Start();
    Console.ReadLine();
  }

  private static void ChangeColor()
  {
    ConsoleColor[] color = { ConsoleColor.Blue, ConsoleColor.Red };
    int n = 0;
    while (true)
    {
      Console.BackgroundColor = color[n = (n + 1) % 2];
      Thread.Sleep(1000);
    }
  }
}
```

In particular, note the method *ChangeColor()* as the color-changing method on the screen. The method is performed in its own thread, and it is thus performed in parallel next to the two print methods. The method is started as a background thread, so that it is terminated as soon as the two print methods are completed.

Her skal du især bemærke metoden *ChangeColor()* som den metode, der skifter farve på skærmen. Metoden udføres i sin egen tråd, og den udføres således parallelt ved siden af de to print metoder. Metoden er startet som en baggrunds tråd, således at den nedlægges straks de to print metoder er færdige.



3.2 THREAD SAFE OPERATIONS

Many of the basic operations such as assignment, comparison, etc. is not thread safe. That is they can be interrupted by the operating system before they are fully executed, and they can therefore in special cases cause concurrency problems. Instead of defining critical regions with a lock or monitor itself, .NET has an object called *Interlock*, which offers some basic operations that are performed atomic (indivisible).

As an example the following program (*AtomicOperations*) defines two variables of the type *int*. The program has a method is called *Swap()* to swap the two variables, but before a lock is set so that the method is executed thread safe. This may be necessary as the method works on variables defined outside of *Main()* and the two variables could otherwise be changed by other threads.

```
class Program
{
  private static int number1 = 2;
  private static int number2 = 3;

  static void Main(string[] args)
  {
    Console.WriteLine("{0} {1}", number1, number2);
    object o = new object();
    lock (o)
    {
      Swap(ref number1, ref number2);
    }
    Console.WriteLine("{0} {1}", number1, number2);
    number2 = Interlocked.Exchange(ref number1, number2);
    Console.WriteLine("{0} {1}", number1, number2);
    Interlocked.Increment(ref number1);
```

```
    Interlocked.Decrement(ref number2);
    Console.WriteLine("{0} {1}", number1, number2);
    Interlocked.CompareExchange(ref number1, 2, 3);
    Console.WriteLine("{0} {1}", number1, number2);
  }

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

However, you can also swap the two variables in a thread safe way with an atomic operation:

```
number2 = Interlocked.Exchange(ref number1, number2);
```

The method copy *number2* to *number1* and return the old value of *number1*, but it happens atomic, and the result is a *thread safe* swap.

The operations ++ and -- are also indivisible, and if you want to make sure that they are performed atomic, they can be done as follows:

```
Interlocked.Increment(ref number1);
Interlocked.Decrement(ref number2);
```

As a last example will the following operation assign *number1* the value 2 if the value before was 3, but in a thread safe way:

```
Interlocked.CompareExchange(ref number1, 2, 3);
```

3.3 PRODUCER / CONSUMER PROBLEM

A classic example of a concurrency problem is the so-called producer / consumer problem:

1. A thread - the producer - writes (produces) to a buffer (variable).
2. Another thread - the consumer - prints (consumes) the contents of the buffer

If the two threads are not synchronized one may come across

1. that the producer overwrites the buffer before the consumer has processed the content - data is lost
2. that the consumer processes the buffer before the producer has updated it - the consumer processes the same data several times

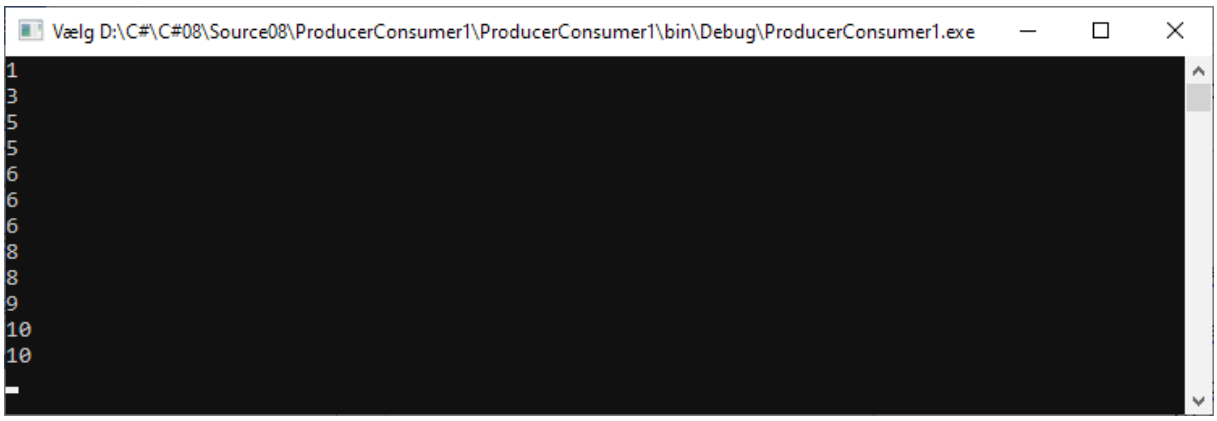
In this section I will show some examples on the producer / consumer problem and how to solve the problem.

In the first example the buffer is a shared variable and the producer overwrites the variable 10 times, but each time the variable is updated, the producer performs a delay of max 1 second. The consumer reads the value of the variable and prints it on the screen, but before reading, it performs a delay of max 1 second.

```
namespace ProducerConsumer1
{
  class Program
  {
    public static readonly Random rand = new Random();
    private static int shared = 0;

    static void Main(string[] args)
    {
      Thread p = new Thread(new ThreadStart(Producer));
      Thread c = new Thread(new ThreadStart(Consumer));
      p.Start();
      c.Start();
      p.Join();
      Thread.Sleep(1000);
      c.Abort();
      Console.ReadLine();
    }
```

```
    private static void Producer()
    {
      for (int i = 0; i < 10; ++i)
      {
        Thread.Sleep(rand.Next(1000));
        shared = i + 1;
      }
    }

    private static void Consumer()
    {
      while (true)
      {
        Thread.Sleep(rand.Next(1000));
        Console.WriteLine(shared);
      }
    }
  }
}
```

If you run the program the result could be:

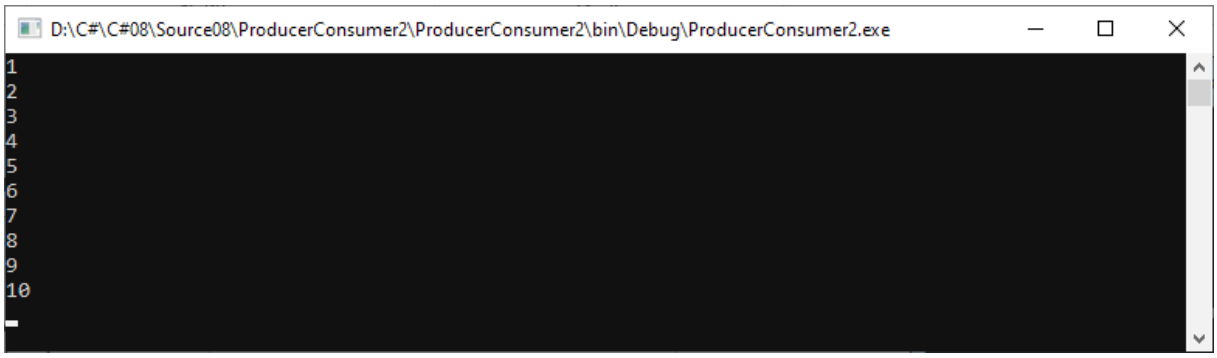


The producer and the consumer are executed in each their thread, but without being synchronized. The result is that some of the produces updates are lost, while the consumer read the same value more times. You should note that the primary thread joins the producer to wait for the producer to finish, and when it happens the primary thread is suspended in 1 second to give the consumer a chance to read the last update. Also note how to abort a running thread.

The next example is exactly the same example but this time the producer and consumer are synchronized using a *lock*:

```
namespace ProducerConsumer2
{
  class Program
  {
    public static readonly Random rand = new Random();
    private static int shared = 0;
    private static bool changed = false;
    private static object binLock = new object();

    static void Main(string[] args)
    {
      Thread p = new Thread(new ThreadStart(Producer));
      Thread c = new Thread(new ThreadStart(Consumer));
      p.Start();
      c.Start();
      p.Join();
      Thread.Sleep(1000);
      c.Abort();
      Console.ReadLine();
    }

    private static void Producer()
    {
      for (int i = 0; i < 10;)
      {
        Thread.Sleep(rand.Next(1000));
        if (!changed)
        {
          lock (binLock)
          {
            shared = ++i;
            changed = true;
          }
        }
      }
    }

    private static void Consumer()
    {
      while (true)
      {
```

```
        Thread.Sleep(rand.Next(1000));
        if (changed)
        {
          lock (binLock)
          {
            Console.WriteLine(shared);
            changed = false;
          }
        }
      }
    }
  }
}
```



The actual synchronization is done using a variable *changed*, which both producer and consumer can see. The variable is set to *false* from the start. This means that the consumer will not read, whereas the producer will update. The synchronization lies in how this variable changes, and for both producer and consumer it happens in a critical region.

If you see it all from the producer, it runs a loop where it updates the variable *shared* 10 times. For each iteration, it tests the variable *changed*. If it is *false* corresponding to that *shared* may be updated, the producer tries to get the lock, after which it updates and sets *changed* to *true*. Note that the consumer will only try to read (and get the lock) after the producer has updated *shared* and *changed*. From the consumer's point of view, everything is the opposite, and it will only try to get the lock if the *change* is *true*.

EXERCISE 4: PRIME GENERATOR

You must write a console application which prints the sequence of primes on the screen, when the first prime (2) has index 1. The program must be written as a thread (a producer) that generates the primes and another thread (a consumer) that prints the primes (and there index). The two threads must be synchronized such the consumer prints all primes (and each prime only one time). From start the time to determine a prime is short, but if the program is running for a long time you should observe that the consumer has to wait for the producer.

When the two threads are started *Main()* should enter a loop where to wait for the user to enter a key:

```
static void Main(string[] args)
{
  (new Thread(() => Producer())).Start();
  (new Thread(() => Consumer())).Start();
  for (ConsoleKeyInfo key = Console.ReadKey(true); ; key = Console.
  ReadKey(true))
  {
    if (key.Key == ConsoleKey.Escape) running = false;
    else if (key.Key == ConsoleKey.Enter) running = true;
  }
}
```

The result is that it the user keys *Escape* a variable *running* is set to *false* and if the users keys *Enter* the variable is set to *true*. If *running* is *false* the consumer should be inactive, but the producer must continue to generate primes, but if *running* is *true* the producer must wait for the consumer to read the generated prime.

The program should run in an infinity loop. If the program running long enough it cannot generate more primes (an *ulong* has max capacity). The greatest prime in an *ulong* is 18446744073709551557, and when you reach this prime the program should basically wrap around and start from 2 again. I do not think you will ever get in that situation!

3.4 MORE CONSUMERS

In this section I will show an example with 1 producer that inserts 300 numbers into a buffer, but there are 10 consumers who all read from the same buffer. The following must be handled:

1. The producer must not insert in a full buffer.
2. A consumer must not read from an empty buffer.
3. Mutual exclusion must be implemented for producer and consumer, so that only one can operate on the buffer at a time.

The last problem I want to solve with a mutex so that it is only the thread that has the mutex object that can use the buffer.

The success criterion (for the solution to the simultaneity problem) is that all consumers come to, and that all numbers are read, and that none of the numbers are read twice.

I start with a class *Buffer* which is implemented as a circular buffer. The size of the buffer and thus how many elements it must fit should be defined by the constructor. The class should be thread safe so each method trying to take a lock before performing its operation. This way, you ensure that there is only one thread at a time that can modify the buffer, but note that it is not necessarily a good idea to make a class thread safe that way, as it assign overhead to the individual methods, and this may affect class's performance. Also note, that the individual methods has a lock does not ensure that buffer is used correct. For example a thread can test that the buffer is non empty, be interrupted and another thread can remove an element from the buffer which make the buffer empty. When the first thread then try to remove an element, the buffer is empty which result in an error. In this case, it is not used that the class's methods are thread safe, as the concurrency control is located where the class is used.

```
namespace ProducerConsumer3
{
  public class Buffer<T>
  {
    private object lockObj = new object();
    private T[] buff;           // array to the buffer
    private int head = 0;      // index to the place in front of the first
                                  element
    private int tail = 0;      // index of the last element of the buffer
    private int count = 0;    // number of elements in the buffer

    public Buffer(int n)
    {
      lock (lockObj)
      {
        buff = new T[n];
      }
    }

    public int Count
    {
      get
      {
        lock (lockObj)
        {
          return count;
        }
      }
    }

    public bool IsEmpty
    {
      get
      {
        lock (lockObj)
        {
          return count == 0;
        }
      }
    }
```

```
public bool IsFull
{
  get
  {
    lock (lockObj)
    {
      return count == buff.Length;
    }
  }
}

public T Peek()
{
  lock (lockObj)
  {
    if (IsEmpty) throw new Exception("The buffer is empty");
    return buff[head];
  }
}

public T Remove()
{
  lock (lockObj)
  {
    if (IsEmpty) throw new Exception("The buffer is empty");
    T elem = buff[head];
    head = Next(head);
    --count;
    return elem;
  }
}

public void Insert(T elem)
{
  lock (lockObj)
  {
    if (IsFull) throw new Exception("The buffer is full");
    buff[tail] = elem;
    tail = Next(tail);
    ++count;
  }
```

```
    }

    private int Next(int n)
    {
      return (n + 1) % buff.Length;
    }
  }
}
```

The methods of the class are generally simple and hardly require special comments. With the class *Buffer* ready the program can be written as:

```
namespace ProducerConsumer3
{
  class Program
  {
    public static readonly Random rand = new Random();
    private static Buffer<int> buffer = new Buffer<int>(10);
    private static Mutex mutex = new Mutex();
    private static int[] count = new int[10];

    static void Main(string[] args)
    {
      Thread p = new Thread(new ThreadStart(Producer));
      p.Start();
      Thread[] c = new Thread[count.Length];
      for (int i = 0; i < count.Length; ++i)
        (c[i] = new Thread(new ParameterizedThreadStart(Consumer))).
         Start(new Param { Index = i, Producer = p });
      for (int i = 0; i < c.Length; ++i) c[i].Join();
      Console.WriteLine();
      for (int i = 0; i < count.Length; ++i) Console.Write("{0, 3}",
      count[i]);
      Console.WriteLine();
      Console.ReadLine();
    }

    private static void Producer()
    {
      for (int n = 1; n <= 300;)
      {
        Thread.Sleep(rand.Next(100));
```

```
        mutex.WaitOne();
        try
        {
          if (!buffer.IsFull) buffer.Insert(n++);
        }
        finally
        {
          mutex.ReleaseMutex();
        }
      }
    }

    private static void Consumer(object data)
    {
      Param param = (Param)data;
      int id = Thread.CurrentThread.ManagedThreadId;
      while (true)
      {
        Thread.Sleep(rand.Next(1000));
        mutex.WaitOne();
        try
        {
          if (!buffer.IsEmpty)
            Console.Write("{0, 2}:{1, 4} ", id, buffer.Remove());
          ++count[param.Index];
        }
        finally
        {
          mutex.ReleaseMutex();
        }
        if (!param.Producer.IsAlive) break;
      }
    }

    class Param
    {
      public int Index { get; set; }
      public Thread Producer { get; set; }
    }
  }
}
```

Main() creates and start the producer. This time the consumer has parameters, which are an index and the producer thread. The index is used to let a consumer count how many times it read from the buffer and the producer thread is used to let the consumer test where the

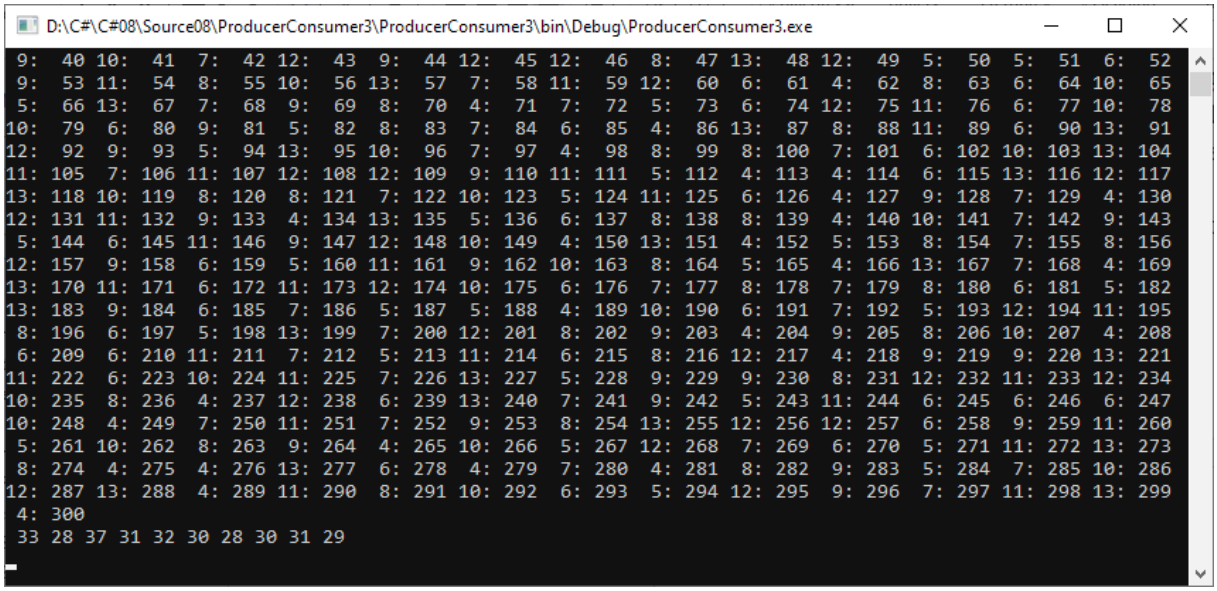
producer still is running. To transfer these parameters to the consumers is defined an inner class *Param*. When all consumers are started the primary thread joins the consumers to wait until they all are finished, when the program prints the number of times the individual consumers has read the buffer. These 10 numbers should all be around 30 indicating that all consumers come to.

This time a mutex is used for the concurrency control. A mutex is a standard Windows synchronization mechanism, which is a binary semaphore. There is only one thread that can have a mutex, and if another thread tries to get the same mutex, the thread will be blocked until the first thread releases the mutex object. It is important to be aware that a mutex is a good candidate for a deadlock situation if for some reason it is not released.

If you look at the producer, it adds elements to the buffer if it is not full. Before the producer updates the buffer, it tries to get the mutex object with a *WaitOne()*. If this is not possible, the thread is suspended and it has to wait. If it gets the mutex object, other threads trying to get the same mutex will have to wait until this thread releases the mutex object. After the producer gets the mutex object, it will update the buffer if it is not full. This operation is placed in a try block, so that the subsequent *final* block is performed with certainty, and one is thus sure that the producer releases the mutex object.

The consumer works in the same way in principle, only a consumer will instead remove an element from the buffer. It will try to get the mutex object, and if it succeeds, it will remove an item from the buffer if it is not empty. Note that while the consumer is trying to remove an item and print it on the screen, the producer (and other consumers) cannot update the buffer due to the mutex object.

If you run the program the result could be:

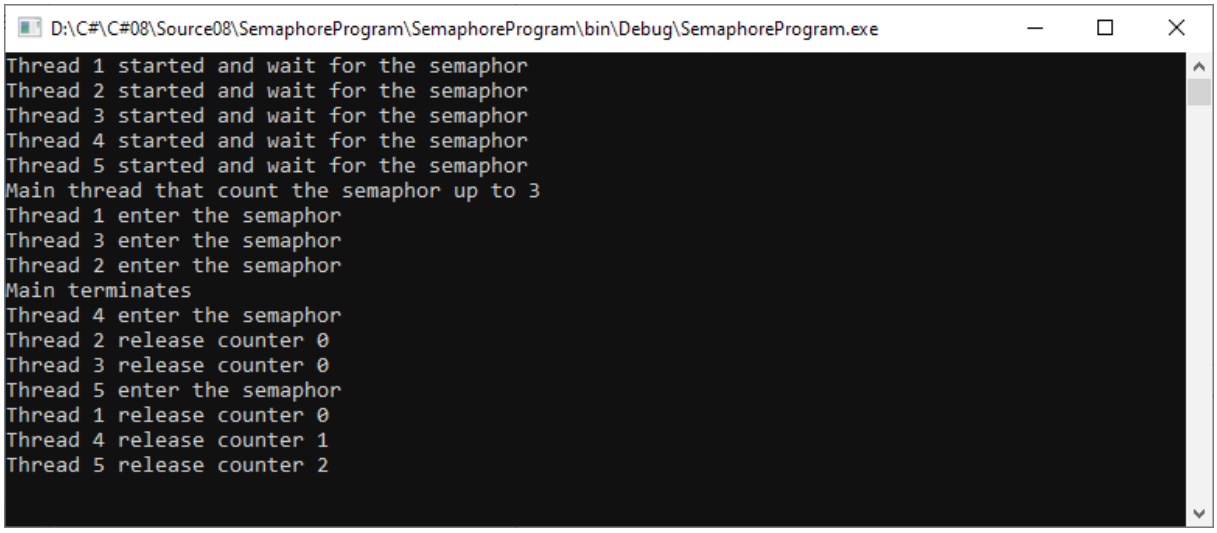


3.5 A SEMAPHORE

A semaphore (also called a count semaphore) is another classic synchronization mechanism used to ensure that a maximum number of threads can access a resource. In the following example, it is a semaphore that counts from 0 to 3 and it starts with being 0. A thread can get the semaphore if its value is greater than 0. Otherwise it has to wait. When a thread gets the semaphore, it counts down the semaphore by 1 and will thus possibly block other threads from getting the semaphore. A thread releases the semaphore by counting it 1 up. Since this semaphore can have a maximum of 3, there are a maximum of 3 threads that can count the semaphore down, and there are thus a maximum of 3 threads that can have the semaphore at the same time.

The example (*SemaphoreProgram*) starts 5 threads, where each thread performs some work on the processor. The example should show that due to the semaphore, only 3 threads can be added.

```
class Program
{
  public static readonly Random rand = new Random();
  private static Semaphore semaphor = new Semaphore(0, 3);

  static void Main(string[] args)
  {
    for (int i = 1; i <= 5; i++)
      (new Thread(new ParameterizedThreadStart(Worker))).Start(i);
    Thread.Sleep(1000);
    Console.WriteLine("Main thread that count the semaphor up to 3");
    semaphor.Release(3);
    Thread.Sleep(1000);
    Console.WriteLine("Main terminerer");
  }

  private static void Worker(object data)
  {
    int id = (int)data;
    Console.WriteLine("Thread {0} started and wait for the semaphor", id);
    semaphor.WaitOne();
    Console.WriteLine("Thread {0} enter the semaphor", id);
    int n = rand.Next(5) + 1;
    while (n-- > 0) for (uint u = 0; u < 20000000; ++u)
      Math.Sin(Math.Log((Math.Sqrt(n))));
    Console.WriteLine("Thread {0} release counter {1}", id, semaphor.
    Release());
  }
}
```



The program creates a semaphore, which can count from 0 to 3. In *Main()*, 5 threads are created, all of which perform the method *Worker()*. When a thread is started, a number is transmitted. This is just so that the thread can print the number so that you can see on the screen which thread is active.

After the threads are created and started, none of them do anything. They are all waiting for the semaphore, which is 0. In *Main()*, the semaphore is counted up by 3, which means that there are now three threads that can get the semaphore and start their work. The last two threads appear when one of the first three is finished and counts the semaphore.

The method *Worker()* should only illustrate a thread that performs a job that here consists of performing a large number of calculations that take a long time overall. After the method prints a message, it waits for the semaphore. It can get the semaphore if its value is greater than 0, and when it does, the semaphore is automatically counted down using the method *WaitOne()* and the thread continues. If this results in the semaphore becoming 0, then other threads have to wait for the semaphore to be positive again. It is the responsibility of this thread to do this by performing a *Release()*.

EXERCISE 5: STORAGE PROGRAM

Create a console application project that you can call *StorageProgram*. The program is an example of using both a *mutex* and a *semaphore*.

Start by writing the following class, which is an encapsulating of a *Dictionary<int, string>*, when it is a requirement that the class must be written as a singleton:

```
class Storage
{
  private static int ID = 0; // the consecutive numbering of objects
  private Dictionary<int, string> cache = new Dictionary<int, string>();

  public int Insert(string name)
  {
    // add a new name to the data structure identified by the next id
    // that is has the next id as key
  }

  // returns a name with an id or null, the id is not found
  public string this[int id] { get; }

  // Returns the number of names
  public int Length { get; }
}
```

The class should illustrate a data structure consisting of names, identified by a sequential number.

The goal of the exercise is to create threads that reads names from the data structure as well as threads that adds names (performing *Insert()*). It should be such that up to 5 threads simultaneously read, while only a single thread can add names. It must be controlled by a mutex and a semaphore. To make things a little easier, you should encapsulate the two synchronization objects in the below class when it is a requirement that the class must be written as a singleton:

```
class StorageLock
{
  private Mutex writeLock = new Mutex();
  private Semaphore readLock = new Semaphore(5, 5);

  public void SetWriteLock()
  {
    writeLock.WaitOne();
  }

  public void ReleaseWriteLock()
  {
    writeLock.ReleaseMutex();
  }
```

```
  public void SetReadLock()
  {
    readLock.WaitOne();
  }

  public void ReleaseReadLock()
  {
    readLock.Release();
  }
}
```

As a next step you must write a class *Writer* that can add names to the data structure *Storage*:

```
class Writer
{
private String[] names;

  public Writer(String[] names)
  {
    this.names = names;
  }

  public void Write()
  {
    foreach (string name in names)
    {
      // set a write lock
      // add name
      // release the lock
      // sleep a random time less than 2 seconds
    }
  }
}
```

The class should in the constructor start the method *Write()* as a thread.

You must corresponding write a class *Reader* that can read the names in the data structure *Storage*:

```
class Reader
{
  public void Read()
  {
    while (true)
    {
      // set a read lock
      // read a name with a random id
      // release the lock
      // prints the name, if it is not null
      // sleep a random time less than a ½ second
    }
  }
}
```

The class must in the constructor start *Read()* as a background thread.

Then there is the only main program:

```
public class StorageProgram
{
  private static final string[] boys = ...  // array with 10 boy names
  private static final string[] girls = ... // array with 10 girl names

  public static void Main(string[] args)
  {
    // creates a thread to a Writer object which adds boy names
    // creates a thread to a Writer object which adds girl names
    // Creates 10 Reader objects
    // the primary thread should join the two writer threads
  }
}
```

Test the program from a prompt and check that everything works as intended.

EXERCISE 6: DEADLOCK

Synchronization and locks means that threads are blocked and put in a waiting position, and it may lead to deadlock. It can happen if two threads must use two shared resources A and B. If one thread puts a lock on A, while the other thread puts a lock on B, and if the first thread then tries to get the lock on B (what it cannot), and the second thread attempts to get the lock on A (what it cannot), there is a deadlock. As an example is below shown a program which provoke a deadlock and the program "hangs":

```
namespace Deadlock
{
  class Program
  {
    private static object lock1 = new object();
    private static object lock2 = new object();

    private static int count = 0;

    static void Main(string[] args)
    {
      (new Thread(() => Run1())).Start();
      (new Thread(() => Run2())).Start();
    }

    public static void Run1()
    {
      while (true)
      {
        // perform Work1()
        // sleep in some milliseconds
      }
    }

    public static void Run2()
    {
      while (true)
      {
        // perform Work2()
        // sleep in some milliseconds
      }
    }
```

```
    public static void Work1()
    {
      // set a lock on lock1
      // set a lock on lock2
      // increase count
      // print the thread id and the value of count
      // release the locks
    }

    public static void Work2()
    {
      // set a lock on lock2
      // set a lock on lock1
      // increase count
      // print the thread id and the value of count
      // release the locks
    }
  }
}
```

Test the program, and you should observe that the program hangs. Some times the program hangs quickly and some time the program can run for a long time.

4 A TIMER

A timer is a thread that at certain time intervals calls a method, which is usually called a timer callback method or just a timer function. It is a *void* method with a single parameter of the type *object*. The timer function is performed in the thread that runs the timer.

This example starts a timer, which ticks every second. That is the timer function is performed every second and it prints a message on the screen:

```
namespace ATimer
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Primary thread: {0}",
        Thread.CurrentThread.ManagedThreadId);
      TimerCallback action = new TimerCallback(PrintTime);
      Timer t = new Timer(action, "The timer name", 2000, 1000);
      while (true) ;
    }

    private static void PrintTime(object state)
    {
      Console.WriteLine("Thread: {0}\nThe time: {1}, An important
      message: {2}",
        Thread.CurrentThread.ManagedThreadId, DateTime.Now.
        ToLongTimeString(),
        state.ToString());
    }
  }
}
```

TimerCallback is a delegate that can reference a timer function. A timer function is always a *void* method with an object as a parameter. In this case, the variable action refers to the timer function *PrintTime()*. To start the timer, create a new *Timer* object:

- the first parameter is the timer function
- the next is a parameter to the timer function and should simply be an object
- the third parameter indicates how many milliseconds to go before the timer ticks for the first time
- the last parameter indicates how often the timer ticks measured in milliseconds

5 WPF AND THREADS

The purpose of a program with a graphical user interface is that the program must be able to interact with the user at all times. There must be no breaks where the user cannot do anything until the program has completed some time consuming action. The solution is of course several threads, but the use of several threads that want to update the user interface is not without problems. Hence this chapter.

All WPF applications start two threads, one of which has the task of rendering (drawing) the user interface, while the other is responsible for the user interaction. Here the first is hidden and runs in the background. The second thread is called the UI thread and is the one with which the user interacts. WPF requires that its UI objects be attached to the UI thread, and an object can only be applied to the thread where it is created. If you try to use the object from another thread, you get a runtime exception and the program crashes. This presents a problem if another thread wants to update the user interface, and this problem is handled by a so-called *Dispatcher* class, which is a kind of priority message loop for WPF applications. A WPF application typically has a single *Dispatcher* object for a single UI thread (although there may be several) through which all user interaction is passed. In this context, prioritization is important so that certain operations are performed before others. For example the rendering mechanism has higher priority than the input system to ensure that e.g. animations are still active no matter what the user does with the mouse and keyboard.

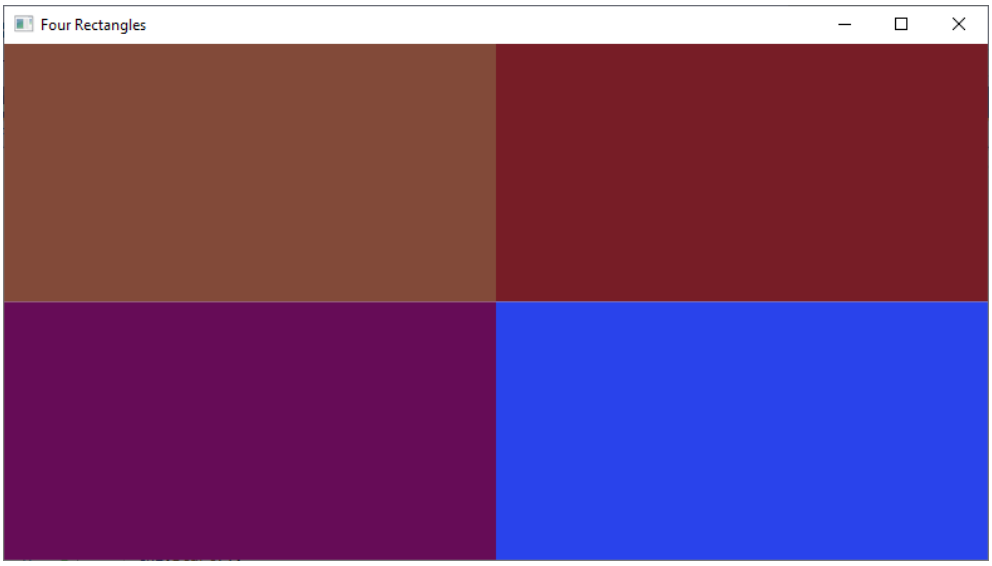
The sum of all this is that any attempt to update the user interface from a secondary thread will fail, but unless one delegates the task to the *Dispatcher* object.

The class *DispatcherObject* inherits directly from *Object*, and the class is the indirect base class for most WPF components. It has several important tasks, including providing access to the *Dispatcher* object to which a given UI object is associated.

So you cannot update a UI control from a thread that does not own the control, and if you try to do something with the control from another thread, you get a runtime error. If, for example you consider the following program

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    Label name = new Label();
    name.Content = "Knud";
    Thread thread = new Thread(
      new ThreadStart(delegate() { name.Content = "Svend" }));
    thread.Start();
  }
}
```

it cannot run. You get a runtime error when a secondary thread tries to update a property on the *Label* component that is owned (created) by another thread. It is necessary to use the *Dispatcher*. To show how the following example opens a window showing four rectangles:



The program starts 4 threads in addition to the two default threads, and each of the four threads updates the user interface. It is simple and consists only of a *Grid* with 4 cells, where each cell contains a rectangle:

```
<UniformGrid Rows="2" Columns="2">
  <Rectangle Name="rect1" />
  <Rectangle Name="rect2" />
  <Rectangle Name="rect3" />
  <Rectangle Name="rect4" />
</UniformGrid>
```

If you run the program, you get a window as shown above, but the color of the rectangles changes randomly by approx. one second apart. This is done by starting 4 threads, and each thread then updates one of the four rectangles:

```
public partial class MainWindow : Window
{
  private static Random rand = new Random();
  private Timer timer1;
  private DispatcherTimer timer2 = new DispatcherTimer();

  public MainWindow()
  {
    InitializeComponent();
    StartThreads();
    StartTimers();
  }

  private void StartThreads()
  {
    Thread th1 = new Thread(ThreadFunc);
    Thread th2 = new Thread(
      delegate()
      {
        while (true)
        {
          Thread.Sleep(rand.Next(500, 1000));
          Dispatcher.Invoke(
            new Action(delegate() { rect2.Fill = CreateBrush(); } ) );
        };
      });
    th1.IsBackground = true;
    th2.IsBackground = true;
    th1.Start();
    th2.Start();
  }

  private void StartTimers()
  {
    timer1 = new Timer(UpdateRect, null, 1000, 1500);
    timer2.Interval = TimeSpan.FromMilliseconds(1000);
    timer2.Tick += new EventHandler(
           delegate(object s, EventArgs a) { rect4.Fill =
           CreateBrush(); });
    timer2.Start();
  }
```

```
  private void ThreadFunc()
  {
    while (true)
    {
      Thread.Sleep(rand.Next(1000, 2000));
      Dispatcher.Invoke(new Action(UpdateRect));
    }
  }

  private void UpdateRect()
  {
    rect1.Fill = CreateBrush();
  }

  private void UpdateRect(object state)
  {
    Dispatcher.Invoke(new Action(delegate() { rect3.Fill =
    CreateBrush(); }));
  }

  private Brush CreateBrush()
  {
    return new SolidColorBrush(Color.FromRgb((byte)rand.Next(256),
        (byte)rand.Next(256), (byte)rand.Next(256)));
  }
}
```

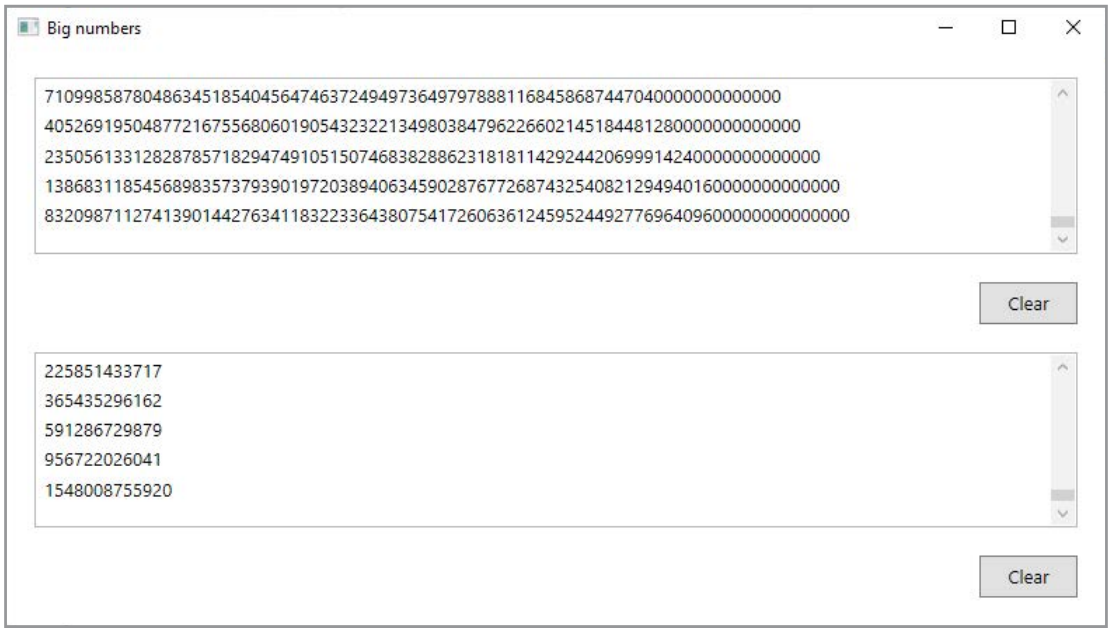
First, note the last method, which creates a brush with a random color. Next, notice the method *ThreadFunc()*, which runs in an infinite loop and every 1 - 2 seconds updates the upper left rectangle by calling the first *UpdateRect()* method. The method *ThreadFunc()* is executed in its own thread, and since it must update a UI element with the method *UpdateRect()*, it must be executed through the dispatcher.

The thread is started in the method *StartThreads()*, and it starts another thread, which updates the top right rectangle. It happens in exactly the same way and it is again necessary to update the rectangle through the dispatcher. The only difference is that this time the thread function is written inline as an anonymous method.

At the beginning of the program is defined a timer. It ticks every 1½ seconds, calling the lower *UpdateRect()* method, which updates the lower left rectangle. A timer starts a thread and it must therefore update the user interface through the dispatcher. The timer is started in the method *StartTimers()*, and here another timer is started, with it the type *DispatcherTimer*. The difference is that it contains logic to update WPF components through the dispatcher, and you are thus free to call the dispatcher yourself.

EXERCISE 7: BIG NUMBERS

Write a program which opens the following window with two list boxes and two buttons:



The program must start a *Timer* which ticks every second and insert the next factorial (as a *BinInteger*) in the upper list box, and a *DispatcherTimer* that also ticks every second and insert the next Fibonacci number in the lower list box. The two button should be used to clear the list boxes.

6 LINQ

LINQ stands for *Language Integrated Query* and is an extension to the traditional .NET languages with the aim of making it easier and more uniform to work with various data sources such as usual arrays, collections, database tables, XML documents etc. In many (maybe all) programs, there is a need to go through collections of data and do something with a selection of elements:

- traverse an array using a combination of *for* and *if*
- traverse a collection using an iterator and a comparator
- find data in a database table using a SQL SELECT
- find data in an XML document using *XPath* or *XQuery*

and the purpose of LINQ is to streamline that kind of data processing so that it is performed in the same way, whether the data source is one or the other.

Syntactically, LINQ is similar to SQL, but it is important to note that LINQ is not SQL, and that it is simply a matter that a LINQ expression in many ways are used similar to SQL. In addition to a SQL-like syntax, LINQ builds on (uses) the following program constructs in C#:

- variables with implicit types - that is use of *var*
- object initialization, where objects are assigned values using *set* properties
- lambda expressions
- extension methods
- anonymous methods

and in fact several of these constructs have been introduced for the sake of LINQ.

To learn LINQ, there are some syntax that need to be in place, but the syntax is natural in many ways, and once you have got the basic concepts in place, LINQ is quite easy and intuitive to use, and LINQ can help make the code easier to read and write. However, it should be noted that LINQ is not a substitute for the “old” control structures, but that it is merely an alternative that in many contexts can help to write the code more easily.

6.1 INTRODUCTION TO LINQ EXPRESSIONS

In this section I will present some examples of LINQ expressions, but without including all details. The goal is for you to quickly gain an insight into the basic syntax of LINQ expressions.

A LINQ expression is used to execute a query and extract objects from a data source. Traditionally, such a task will be solved by traversing the particular data source in a loop, and for each element in the form of an if-statement, select whether the element should be included. This operation can be expressed as a single LINQ expression, and the only requirement is that the data source in question must implement the interface *IEnumerable<T>*, what arrays and all standard collection classes in the .NET framework do. The result of a LINQ query is a collection of objects of an anonymous type, where the specific type is determined by the type of the objects that the expression extracts.

The basis is a few new reserved words, as well as a large collection of LINQ operators in the form of extension methods defined in the namespace *System.Linq*. The basic syntax is

from variable *in* data_source
where condition
select objects

but there are a number of extensions, such as *orderby* if you want the result arranged according to a specific criterion. If you compare the expression with a corresponding SQL statement, it has the form

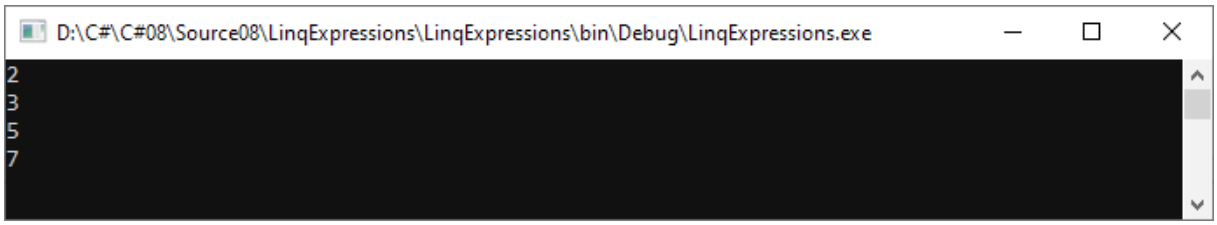
select columns
from table
where condtiton

There are major similarities and the most important difference is that the order has been changed, but it is clear from where the inspiration comes. The reason why the order has been switched has to do with how the built-in intellisense in Visual Studio works.

The easiest approach to LINQ is to consider some examples, and the rest of this section presents LINQ expressions through a series of small examples, including the basic syntax. The program is called *LinqExpressions*.

As an example the following method use a LINQ expression to select all numbers less than 10 in an array and print these numbers on the screen:

```
static void Test01()
{
  int[] numbers = { 11, 7, 19, 3, 17, 2, 5, 13 };
  var res = from t in numbers where t < 10 orderby t select t;
  foreach (var t in res) Console.WriteLine(t);
}
```



The data source is the array of numbers. A data source for a LINQ expression must implement the interface *IEnumerable<T>* (the data source must implement the iterator pattern), which is the case for an array. A LINQ expression can therefore always be applied to an array. The expression itself says that you must select objects *t* from the data source's numbers, but only those objects that are less than 10. Finally, *orderby* defines the elements to be sorted in ascending order. The result is a collection of the selected objects, but the type is anonymous and is generated by the framework. It is rare that it is necessary to know the type, and in most cases it is enough to know that it is possible to iterate over the result with a *foreach* loop. The program uses a loop to print the result.

Note that *orderby* is not necessary (it is not necessary to specify that the items should be arranged). Also note the last *select*, which in turn is needed.

LINQ defines a number of extension methods for arrays and collection classes, and they are defined in the namespace *Sytem.Linq*. Note that Visual Studio automatically inserts a using statement for this namespace. The next example shows how to determine the sum of the numbers in an *int* array with an extension method:

```
static void Test02()
{
  int[] numbers = { 11, 7, 19, 3, 17, 2, 5, 13 };
  int s = numbers.Sum();
  Console.WriteLine(s);
}
```

and if the method is performed it prints the sum 77. There is not much to explain. An extension method *Sum()* is defined for an array, and it is used as a regular instance method. You are encouraged to research what extension methods exists.

The next examples uses a data source consisting of accounts with transactions where a transaction is defined as

```
class Transaction
{
  public double Amount { get; set; }    // the amount for the transaction
  public DateTime Date { get; set; }    // the date for the transaction
}
```

while an account is defined as

```
class Account
{
  public string Phone { get; set; }     // account number as the
                                           owners phone number
  public string Name { get; set; }      // owners name
  public string Job { get; set; }       // owners job name
  public DateTime Date { get; set; }    // owners birthday
  public List<Transaction> Transactions { get; private set; }

  public Account()
  {
    Transactions = new List<Transaction>();
  }

  public Account(string phone, string name, string job, DateTime date)
  {
    Phone = phone;
    Name = name;
    Job = job;
    Date = date;
    Transactions = new List<Transaction>();
  }
}
```

These types are simple model classes. The project also has a class *Accounts* that creates a list with 12 accounts and assign 20 transactions to these accounts:

```
class Accounts
{
  public List<Account> accounts = new List<Account>();

  public Accounts()
  {
    accounts.Add(new Account("12345678", "Svend", "Natmand", new
    DateTime(1960, 3, 25)));
    accounts.Add(new Account("12345679", "Karlo", "Natmand", new
    DateTime(1980, 10, 2)));
    accounts.Add(new Account("12345680", "Alfred", "Skarpretter", new
    DateTime(1993, 1, 5)));
    accounts.Add(new Account("12345681", "Knud", "Skarpretter", new
    DateTime(1985, 12, 24)));
    accounts.Add(new Account("12345682", "Valdemar", "Mestermand", new
    DateTime(1969, 2, 23)));
    accounts.Add(new Account("12345683", "Erik", "Mestermand", new
    DateTime(1971, 8, 27)));
    accounts.Add(new Account("12345684", "Olga", "Klog kone", new
    DateTime(1987, 1, 31)));
    accounts.Add(new Account("12345685", "Gudrun", "Klog kone", new
    DateTime(1961, 11, 15)));
    accounts.Add(new Account("12345686", "Abelone", "Heks", new
    DateTime(1993, 6, 19)));
    accounts.Add(new Account("12345687", "Gunhild", "Heks", new
    DateTime(1979, 4, 7)));
    accounts.Add(new Account("12345688", "Gerda", "Spåkone", new
    DateTime(1973, 10, 5)));
    accounts.Add(new Account("12345689", "Valborg", "Spåkone", new
    DateTime(1983, 9, 23)));
    accounts[0].Transactions.Add(new Transaction { Amount = 20000,
    Date = new DateTime(2020, 3, 23) });
    accounts[0].Transactions.Add(new Transaction { Amount = -1000,
    Date = new DateTime(2020, 12, 3) });
    accounts[1].Transactions.Add(new Transaction { Amount = 25000,
    Date = new DateTime(2020, 1, 2) });
    accounts[2].Transactions.Add(new Transaction { Amount = 30000,
    Date = new DateTime(2019, 12, 24) });
    accounts[2].Transactions.Add(new Transaction { Amount = -35000,
    Date = new DateTime(2020, 2, 29) });
```

```
    accounts[2].Transactions.Add(new Transaction { Amount = 40000,
    Date = new DateTime(2020, 8, 11) });
    accounts[3].Transactions.Add(new Transaction { Amount = 28000,
    Date = new DateTime(2020, 7, 1) });
    accounts[4].Transactions.Add(new Transaction { Amount = 2000, Date
    = new DateTime(2020, 5, 31) });
    accounts[4].Transactions.Add(new Transaction { Amount = 3000, Date
    = new DateTime(2020, 6, 30) });
    accounts[5].Transactions.Add(new Transaction { Amount = 50000,
    Date = new DateTime(2020, 1, 31) });
    accounts[6].Transactions.Add(new Transaction { Amount = 40000,
    Date = new DateTime(2020, 11, 25) });
    accounts[7].Transactions.Add(new Transaction { Amount = 60000,
    Date = new DateTime(2020, 4, 7) });
    accounts[7].Transactions.Add(new Transaction { Amount = -30000,
    Date = new DateTime(2020, 8, 18) });
    accounts[8].Transactions.Add(new Transaction { Amount = 29000,
    Date = new DateTime(2020, 1, 1) });
    accounts[9].Transactions.Add(new Transaction { Amount = 26000,
    Date = new DateTime(2020, 10, 17) });
    accounts[11].Transactions.Add(new Transaction { Amount = 20000,
    Date = new DateTime(2020, 2, 1) });
    accounts[11].Transactions.Add(new Transaction { Amount = 20000,
    Date = new DateTime(2020, 3, 1) });
    accounts[11].Transactions.Add(new Transaction { Amount = 20000,
    Date = new DateTime(2020, 4, 1) });
    accounts[11].Transactions.Add(new Transaction { Amount = 20000,
    Date = new DateTime(2020, 5, 1) });
    accounts[11].Transactions.Add(new Transaction { Amount = 20000,
    Date = new DateTime(2020, 6, 1) });
  }
}
```

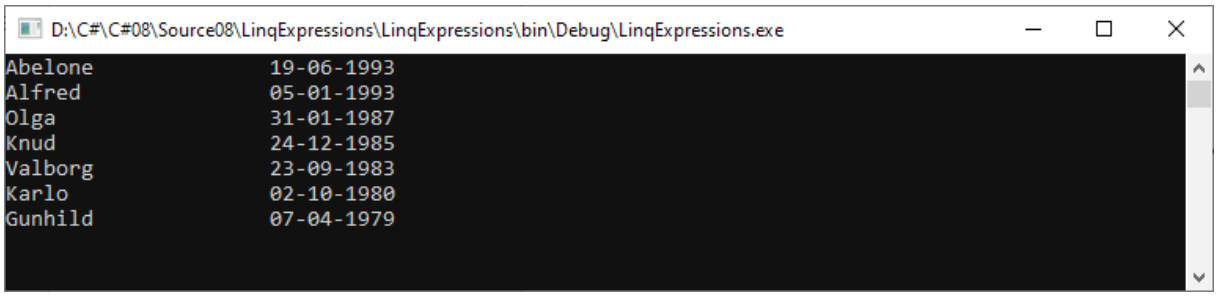
The program *LinqExpressions* has an instance *data* of the class *Accounts* which I in the following examples will use as data source.

The following example prints the name and date of birth of all account holders in a list of accounts where only account holders with an age below 45 must be included, and such that they are arranged by date of birth in descending order:

```
static void Test03()
{
  var query = from k in data.accounts
    where k.Date.AddYears(45) > DateTime.Now
    orderby k.Date descending
    select string.Format("{0,-20}{1}", k.Name, k.Date.ToShortDateString());
  foreach (string s in query) Console.WriteLine(s);
}
```

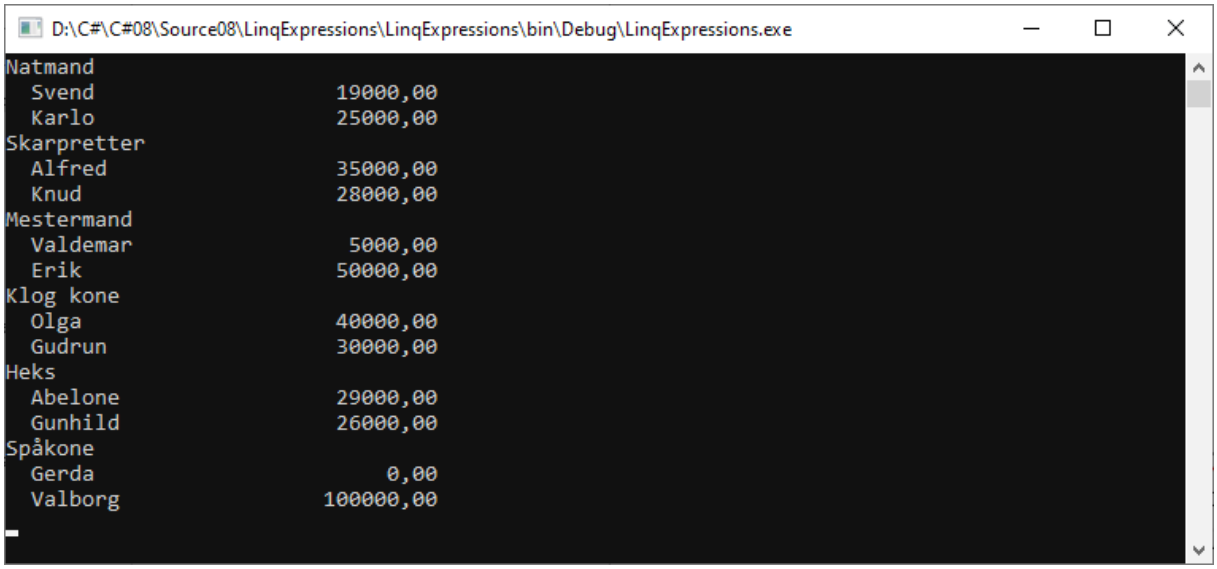
If you look at the LINQ expression, it may not be so easy to guess the syntax and know that the query can be written as shown, but when you see the expression it is easy both to read and understand. The data source is *data.accounts*, which is a list of *Account* objects. From this, some objects called *k* are extracted. The accounts that come with must meet that the date of birth + 45 years must be greater than the date of today, which exactly corresponds to the account holder being under 45 years of age. The selected accounts are arranged by date of birth in descending order, and you should especially note how to indicate that the sorting should be descending. You can also write ascending, which is the default.

In the *select* section, objects of the type *string* are selected, and the result is thus a collection of the type *string*. Note how the individual string objects are created using the *Format()* method from the *string* class.



The next example prints the name and balance of all accounts, but such that they are grouped by job position:

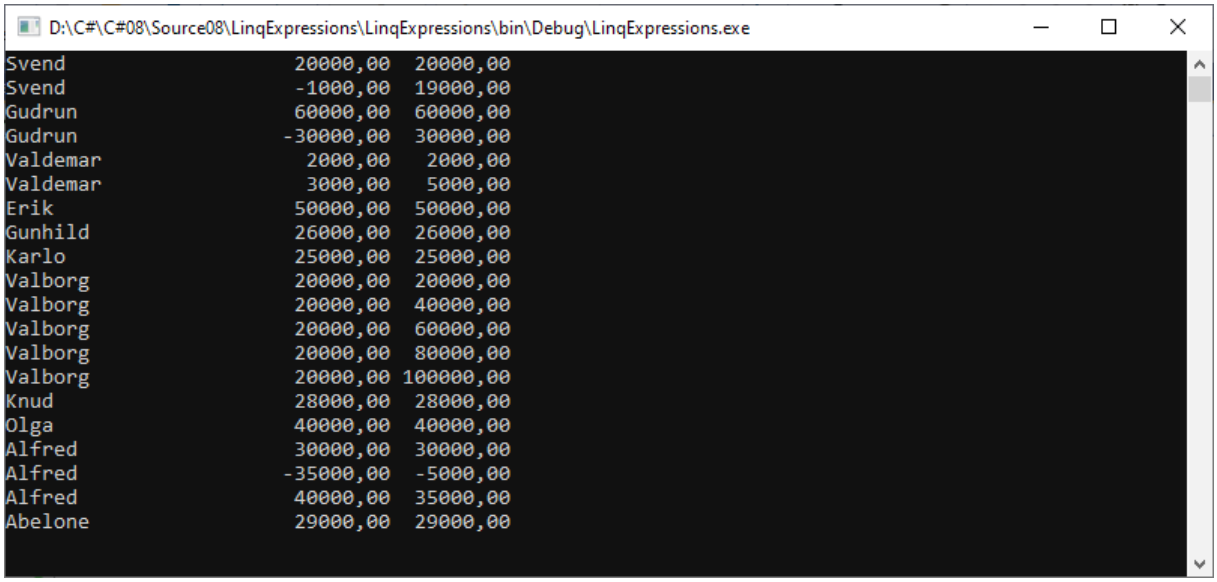
```
static void Test04()
{
  var query = from k in data.accounts group k by k.Job;
  foreach (var group in query)
  {
    Console.WriteLine(group.Key);
    foreach (Account k in group) Console.WriteLine(“ {0, -20}{1,
    12:F2}”, k.Name,
      (from t in k.Transactions select t.Amount).Sum());
  }
}
```



The data source is again *data.accounts*, but then grouped by the property *Job*. The result is a group, where each group has a key, which is a job title (a *string*) as well as a collection of the *Account* objects that have this job title. Note that when printing the result, you run over all groups (keys), and for each group over all accounts in the group. Also note how to determine the balance where to use another LINQ expression which select the amount for all transactions for an account and the sum of these amounts.

The following example prints all transactions, that is the transactions for all accounts where the accounts are arranged after birth date:

```
static void Test05()
{
  foreach (Account k in (from l in data.accounts orderby l.Date select
  l))
    foreach (Transaction t in (from a in k.Transactions orderby a.Date
    select a))
      Console.WriteLine("{0, -20}{1, 12:F2}{2, 10:F2}", k.Name, t.Amount,
        (from s in k.Transactions where s.Date <= t.Date select
        s.Amount).Sum());
}
```



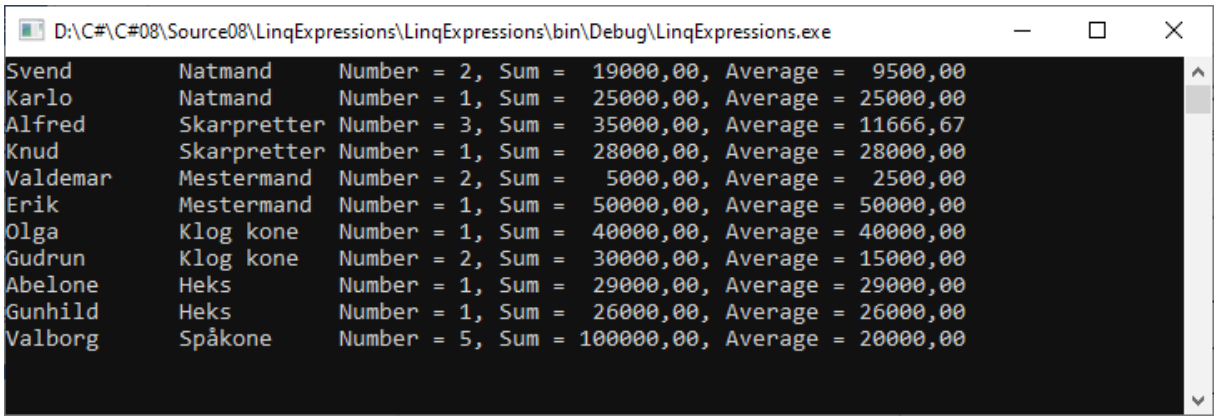
Basically, it's a *foreach* loop over all accounts, but note how I use a LINQ expression to sort the accounts by date of birth. There is also a nested *foreach* loop which for an account loops over all transactions and here again a LINQ expression is used to ensure that the transactions are sorted by date. Finally, a LINQ expression is used to determine for a given account the sum of all transactions up to and including the date of the current transaction. To be completely accurate, it is actually assumed that there is no more than one transaction on the same day.

The aim of the example is to show that with the help of LINQ you can write very powerful statements.

The last example in this section shows a query for all accounts that has transactions and for each account contains

- account holder's name
- account holder's job title
- number of transactions
- the sum of the transactions
- the average of the transactions

```
static void Test06()
{
  var query = from k in data.accounts where k.Transactions.Count > 0
  select new
  {
    k.Name,
    k.Job,
    Count = k.Transactions.Count,
    Avg = k.Transactions.Average(b => b.Amount),
    Total = k.Transactions.Sum(b => b.Amount)
  };
  foreach (var e in query) Console.WriteLine(
    "{0, -12} {1, -11} Number = {2}, Sum = {3, 9:F2}, Average = {4, 8:F2}",
    e.Name, e.Job, e.Count, e.Total, e.Avg);
}
```



The result is a collection of objects of an anonymous type consisting of objects with 5 fields, the last three being the result of extension methods. Here you should notice how these fields are named and how to tell the last two fields what to sum up using a lambda expression. When printing the result, note how to specify the fields in *WriteLine()*.

EXERCISE 8: QUERY AN ARRAY

Write a console application which you can call *QueryProgram*. The program should create and initialize an array:

```
class Program
{
  static int[] numbers = new int[99];

  static void Main(string[] args)
  {
    for (int i = 0; i < numbers.Length; ++i) numbers[i] = i + 1;
```

Write a test method which using a LINQ query extracts and prints a line that includes all numbers in the array divisible by 3. Executes the method from *Main()*.

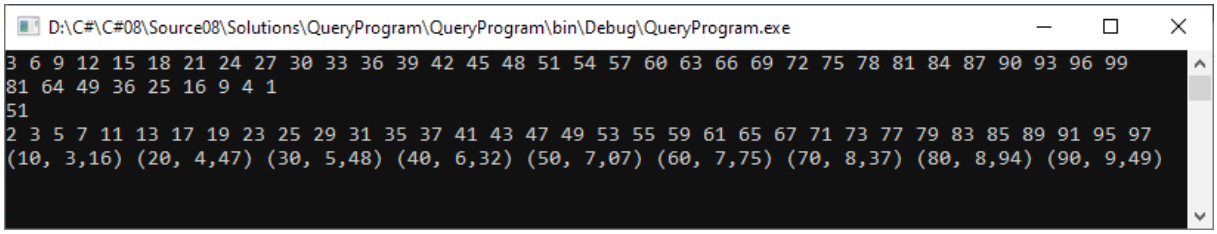
Write another test method which in the same way prints all square numbers but in descending order.

Write a test method which prints the average of all numbers in the array divisible with 3 when you must use a LINQ expression.

Write a test method which using a LINQ query extracts and prints a line that includes all prime numbers in the array. Note that you must write a method to test where a number is a prime.

Write a test method which using a LINQ query creates a collection pairs for all numbers in the array divisible with 10 when a pair must consist of the number and its square root. The method must print the selected pairs on a line.

If you run the program the result should be:



6.2 BASIC SYNTAX

This section describes in a little more detail the basic syntax of a LINQ expression. In fact, there are two ways to write a LINQ expression:

1. Using built-in operators in the language C#, which is usually called the query format.
2. Using extension methods defined in *System.Linq*, commonly called the dot format.

All examples of LINQ expressions in this section can be found in the program *LinqExpressions*.

Given an array:

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

you can using the query format write a LINQ expression where the result is all numbers less than 10 and sorted in ascending order as:

```
var res = from t in numbers where t < 10 orderby t select t;
```

With *from in* you tell from which data source you extract elements, and with *where* you tell which objects from the data source are to be included, with *orderby* how the elements are to be arranged, and with *select* what kind of elements are to be included in the result.

Using the dot format, you can write an expression that gives the same result:

```
var res = numbers.Where(t => t < 10).OrderBy(t => t);
```

numbers is an array that implements the interface *IEnumerable<int>*, and for such an array an extension method *Where()* is defined, which as a parameter has a delegate (here indicated with a lambda expression), which indicates which objects are to be extracted. The result is again a collection, and for this an extension method *OrderBy()* is defined, which as a parameter has a delegate (here indicated with a lambda expression), which indicates according to which values the result must be arranged.

Often the two formats will be alternatives to each other, and the choice has to do with attitudes, but the most important thing is to choose the format that is most readable. However, it is not always possible to choose, and often one can do more with an extension method than is the case with a built-in query expression.

The two formats can also be combined as shown below in an expression that selects all numbers less than 10, so that the numbers are arranged in ascending order and all numbers are different:

```
int[] numbers = { 17, 7, 5, 11, 3, 5, 19, 2, 7, 13, 5 };
var res = (from t in numbers where t < 10 orderby t select t).Distinct();
```

Note that the query expression in parentheses is a collection, and from this collection the extension method *Distinct()* takes the elements that are different.

Formally, the syntax of a query expression is:

```
IEnumerable<T> | T res =
  from id in exp
  [ let id = exp ]
  [ where bool-exp ]
  [ join [ type ] id in exp on exp equals exp [ into id ] ]
  [ orderby crit [ ascending | descending ] ]
  [ group exp by exp [ into id ] ]
  select exp [ into id ]
```

where the square brackets indicate operators that are optional. Then the simplest expression that one can write is something like:

```
var res = from t in numbers select t;
```

which simply selects all elements in the array *numbers*.

Most operators do not require much explanation - at least not if you know SQL - but a few require a mention.

let is used to define a local variable, which is then known and can be used in the rest of the expression. As an example, an expression is shown below which, for all numbers in an array, extracts the square of the numbers deviation from the mean:

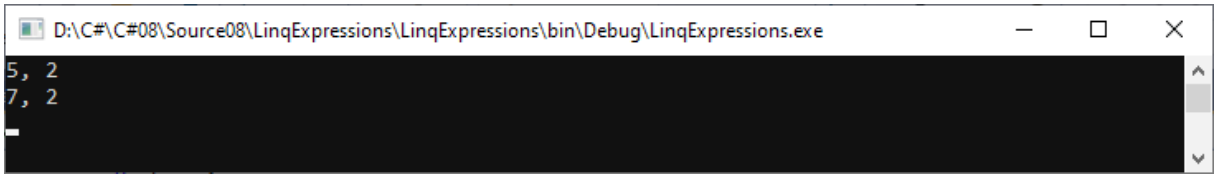
```
private static void Test08()
{
  double[] numbers = { 19, 3, 11, 29, 13, 5, 2, 17, 23 };
  var res = from t in numbers let m = numbers.Average() select Math.
  Pow(t - m, 2);
  foreach (var x in res) Console.WriteLine(x);
}
```

Note that *m* is a local variable whose value is the average of the numbers determined by an extension method.

Also note the operator *into*, which is used to store the result of a query in a variable. The *into* operator has the greatest use in connection with group by:

```
struct Pair
{
  public int Key { get; set; }
  public double Amount { get; set; }
}
private static void Test09()
{
  List<Pair> list = new List<Pair>{ new Pair { Key = 5, Amount = 3.55 },
                                    new Pair { Key = 7, Amount = 13.75 },
                                    new Pair { Key = 7, Amount = 3.15 },
                                    new Pair { Key = 5, Amount = 8.25 } };
  var res = from p in list group p by p.Key into g
            select new { K = g.Key, C = g.Count() };
  foreach (var e in res) Console.WriteLine(e.K + ", " + e.C);
}
```

The list contains four objects of the type *Pair*, where a pair consists of two values. The first part of the LINQ expression extracts a collection of groups grouped by the first field *Key* (there will be two groups) and stores the result in a variable *g*. This variable is used in the select part that creates objects of an anonymous type but consisting of two fields, and in order to initialize these fields, you must be able to refer to the collection that contains the groups. It is therefore necessary to have a name for this collection.



As mentioned, there are two formats for LINQ expressions:

```
int[] numbers = { 17, 5, 11, 3, 19, 2, 7, 13 };
var res = from t in numbers where t < 10 orderby t select t;
var res = numbers.Where(t => t < 10).OrderBy(t => t);
```

where the first is based on operators built into the compiler, while the second is based on extension methods. The difference is that the first is perhaps the most readable, while the second is the most flexible and largely based on delegates and lambda expressions. As mentioned, in most cases it does not matter whether you choose the first or second format, but in reality the compiler will translate the first format to the second.

The where part

LINQ expressions are used to extract a collection from a data source, and usually not all objects in the data source are included. You use *where* followed by a condition to specify which ones to include, and correspondingly you also call *where* for a filter:

```
static void Test10()
{
  string[] names = { "Knud", "Svend", "Valdemar", "Karlo", "Erik", "Albert",
                     "Kurt", "Harald", "Karl" };
  var res = from n in names where n[0] == 'K' select n;
  //      var res = names.Where(n => n[0] == 'K');
  //      var res = names.Where(delegate(string n) { return n[0] ==
          'K'; });
  //      var res = names.Where(IsKing);
  //      var res = names.Where((n, i) => i < 3 || i == 5 || i == 7);
  foreach (string s in res) Console.WriteLine(s);
}
```

that selects all the names that start with a capital K. The same expression can be written as follows with extension methods:

```
var res = names.Where(n => n[0] == 'K');
```

Note in particular that a *select* is not necessary, so it is a fairly simple syntax for selecting objects in a collection. *Where()* is an extension method, and it has one parameter, which is a predicate, which is simply a delegate, which refers to a *bool* method with a parameter of the same type as the data source. Correspondingly, one will usually specify the parameter for *Where()* as a lambda expression. Of course, you can also explicitly state a delegate (a lambda expression is just a short spelling for an anonymous delegate):

```
var res = names.Where(delegate(string n) { return n[0] == 'K'; });
```

If the predicate is more complex, it can be difficult to state the method anonymously, and of course it is not necessary either:

```
var res = names.Where(IsKing);

private static bool IsKing(string name)
{
  string[] kings = { "Knud", "Svend", "Valdemar", "Erik", "Harald" };
  return kings.Contains(name);
}
```

The *Where()* method has an override with another predicate, which in addition to a parameter of the type for the data source also has a parameter, which is an index to the objects of the data source:

```
var res = names.Where((n, i) => i < 3 || i == 5 || i == 7);
```

It may not be used that often, but the opportunity is there.

The result

The result of a LINQ expression is usually a collection of objects of one type or another, and basically there are the following options:

1. A single element whose value is determined by an operator.
2. A collection of type *IEnumerable<T>*, where *T* is the same type as the type of objects in the data source.
3. A collection of type *IEnumerable<T>*, where T is any existing type specified in the *select* section.
4. A collection of the type *IEnumerable<T>*, where *T* is an anonymous type created by the compiler from the *select* part.
5. A collection of the type *IEnumerable<IGrouping<TKey, TElement>>* of grouped objects with a common key.

There are a number of operators - or extension methods - that return a single value, for example

```
int[] numbers = { 19, 3, 11, 29, 13, 5, 2, 17, 23 };
int a = numbers.Sum();
```

which returns the sum of the elements in a data source - provided that the data source contains objects that can be added.

As another example, an expression is shown here which selects all numbers greater than 10 and the smallest of these:

```
int m = numbers.Where(t => t > 10).Min();
```

and below is an expression that tests whether the data source contains a specific object:

```
bool b = numbers.Contains(15);
```

Finally, an expression is shown below which returns the last object in the data source:

```
string[] kings = { "Knud", "Svend", "Valdemar", "Erik", "Harald" };
string s = kings.Last();
```

The following methods returns a number:

- *Aggregate()*
- *Average()*
- *Max()*
- *Min()*
- *Sum()*
- *Count()*
- *LongCount()*

The following methods returns a *bool*:

- *All()*
- *Any()*
- *Contains()*
- *SequenceEqual()*

Finally, the following methods return an object of the same type as the type of data source's objects:

- *ElementAt()*
- *ElementAtOrDefault()*
- *First()*
- *FirstOrDefault()*
- *Last()*
- *LastOrDeafult()*
- *Single()*
- *SingleOrDefault()*
- *DefaultIfEmpty()*

The following expressions should illustrate some of the other return types, all of which are collections. The starting point is the following type:

```
class Person
{
  public string Zipcode { get; set; }
  public string Name { get; set; }
  public string Job { get; set; }

  public override string ToString()
  {
    return string.Format("{0} {1}, {2}", Zipcode, Name, Job);
  }
}
```

and the data source is:

```
Person[] pers = {
  new Person { Zipcode = "7800", Name = "Svend", Job = "King" },
  new Person { Zipcode = "8800", Name = "Frede", Job = "Duke" },
  new Person { Zipcode = "7950", Name = "Knud", Job = "King" },
  new Person { Zipcode = "9990", Name = "Karlo", Job = " Bailiff" },
  new Person { Zipcode = "7500", Name = "Gunner", Job = "Duke" },
  new Person { Zipcode = "7900", Name = "Valdemar", Job = "King" } };
```

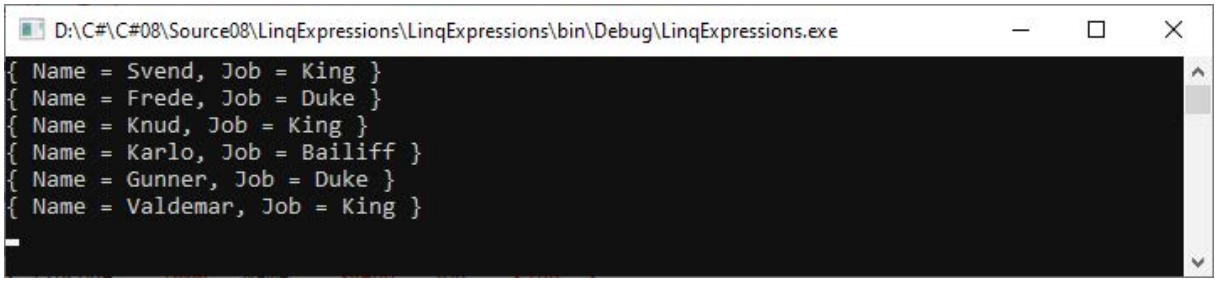
The following expression extracts all persons where the postal code ends in 00:

```
var res = from p in pers where p.Zipcode.Substring(2).Equals("00")
select p;
```

It's not that mysterious, but the important thing here is the return type, which is *IEnumerable<Person>*, as it is the type of objects after *select*. As another example, the following expression extract the name and position of all persons:

```
var res = from p in pers select new { Name = p.Name, Job = p.Job };
```

The type of *res* is *IEnumerable<T>*, where *T* is an anonymous type created by the compiler. The type has no name and is only known inside the method that performs the expression. The type has two fields, both of which are of the type *string* and are called respectively *Name* and *Job*. If you print the result, you get the following:



and here you should especially note how the compiler generated *ToString()* method works.

As another example of an anonymous type, below is an expression that creates a collection of objects of a type that represent a person's zip code and name, but only for persons who are "king":

```
var res =
pers.Where(p => p.Job.Equals("King")).Select(p => new { p.Zipcode, p.
Name });
```

As another example regarding return types for a LINQ expression I will show an expression that returns a collection of groups:

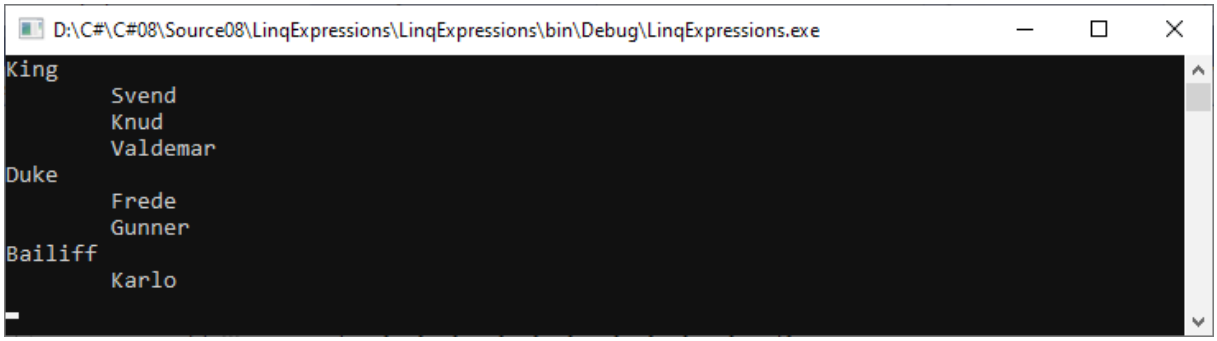
```
var res = pers.GroupBy(p => p.Job);
```

The expression is simple, but the type for the result is this time

IEnumerable<IGrouping<string, Person>>

and as an example, the following code will print the result:

```
foreach (var g in res)
{
  Console.WriteLine(g.Key);
  foreach (var e in g) Console.WriteLine("\t" + e.Name);
}
```



The extension method *Select()* returns a collection of objects of one type or another, and I will also show a variant called *SelectMany()*.

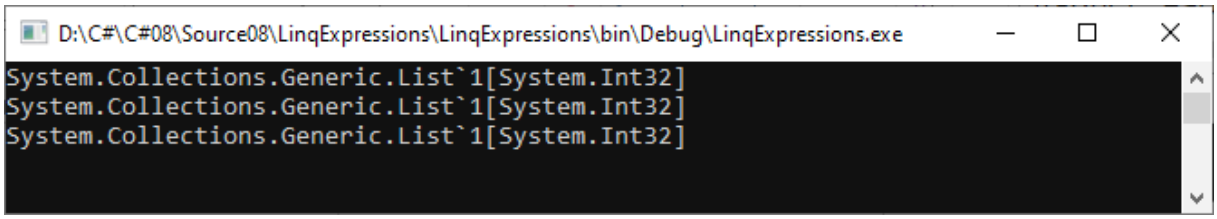
Consider the following data source:

```
List<List<int>> data = new List<List<int>>
{
  new List<int> { 2, 3, 4, 5 },
  new List<int> { 11, 13, 17, 17, 23, 29 },
  new List<int> { 31, 37 }
};
```

which is a list of lists. Then look at the following expression

```
var res = data.Select(obj => obj);
```

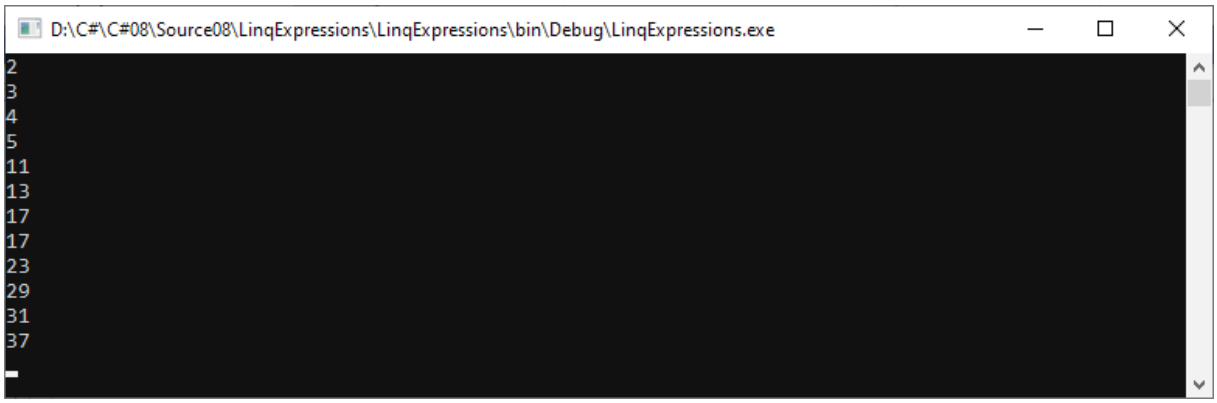
There is not much mystery in the return type that is *IEnumerable<List<int>>*:



It is possible to the next LINQ expression and the result of the expression

```
var res = from v in integers from t in v select t;
```

has the type *IEnumerable<int>* and is thus a collection of integers:



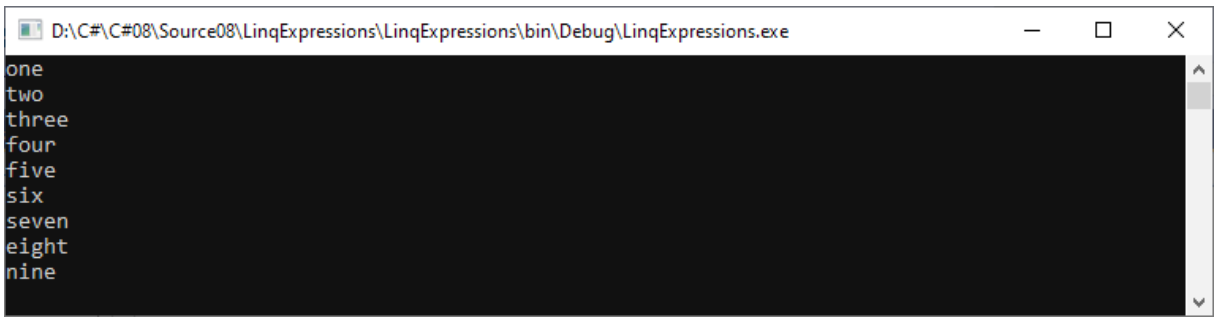
This nested expression can be written simpler with the extension method *SelectMany()*:

```
var res = integers.SelectMany(v => v.Select(t => t));
```

which gives exactly the same result.

As another example, the following expression gives a collection of words:

```
string[] text = { "one two three", "four five six seven", "eight nine" };
var res = text.SelectMany(t => t.Split(' '));
```



Order by

There is an operator *orderby*, which is used to sort the result of a LINQ expression, for example

```
int[] numbers = { 19, 3, 11, 29, 13, 5, 2, 17, 23 };
var res = from n in numbers orderby n select n;
```

where the result is a sorted collection of the type *IOrderedEnumerable<int>*. It is thus simple to perform a sorted query, and the sorting is otherwise done using quicksort. As another example, below is a sorted query, but this time written as an extension method:

```
string[] names = { "File1", "File2", "File10", "letter1", "letter10",
"letter2",
                   "file1", "file2", "file10", "Letter10", "", "Letter2" };
var res = names.OrderBy(s => s);
```

If you look at the result, you can see that it is a sort that does not distinguish between uppercase and lowercase letters, which is the default. In general, you can compare strings in several ways, and the extension method *OrderBy()* has an override, where you can specify a comparator, which tells how objects are to be compared. Here one must pay special attention to the class *StringComparer*, which contains several ready-made comparators for strings. For example the following expression will perform a "regular" sorting, where strings are sorted by code values in the Unicode table:

```
var res = names.OrderBy(s => s, StringComparer.Ordinal);
```

The result is, among other things that the uppercase letters come before the lowercase letters.

Of course, you can also write your own comparator, which is a class that implements the interface *IComparer<T>*. In this case, the names of the texts in the array *names* suggest that it could be file names, and here one could for example be interested in that *letter2* should come before *letter10*. It can be solved with a comparator:

```
class NameOrder : IComparer<string>
{
  public int Compare(string s1, string s2)
  {
    StringComparer cmp = StringComparer.CurrentCultureIgnoreCase;
    if (string.IsNullOrEmpty(s1) || string.IsNullOrEmpty(s2))
      return cmp.Compare(s1, s2);
    int n1 = NumberIndex(s1);
    int n2 = NumberIndex(s2);
    if (n1 < s1.Length && n2 < s2.Length)
    {
      int value = cmp.Compare(s1.Substring(0, n1), s2.Substring(0, n2));
      if (value != 0) return value;
      return Convert.ToDouble(s1.Substring(n1)).CompareTo(
                    Convert.ToDouble(s2.Substring(n2)));
    }
    return cmp.Compare(s1, s2);
  }

  private int NumberIndex(string s)
  {
    int n = s.Length - 1;
    while (n >= 0 && char.IsNumber(s[n])) --n;
    return n + 1;
  }
}
```

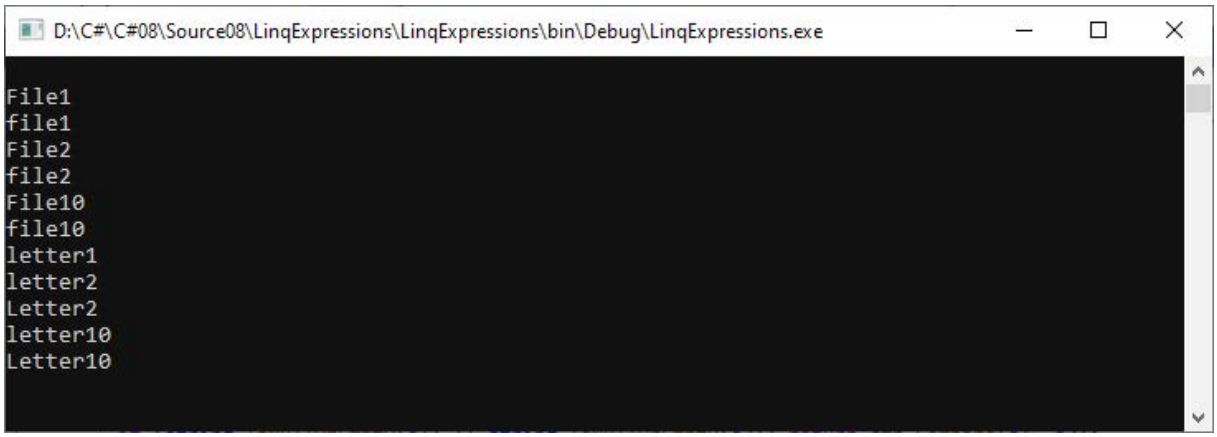
NameOrder is a comparator that compares strings so there is no distinction between uppercase and lowercase letters, and so that the order of the characters corresponding to the current culture is taken into account. In addition, the strings are arranged according to the number value (and not the individual digits as characters) if a string ends with a number.

Note in particular that the comparator implements the interface *IComparer<string>*. In this case, the comparison works as follows:

- First, a standard comparator is defined for strings that do not distinguish between uppercase and lowercase letters and that arrange the characters according to the current culture.
- If one of the strings is *null* or empty, use this comparator.
- If both strings end with a number, first compare the part of the strings that is not a number. If they differ, the result is the default scheme. Otherwise, the scheme is determined by the value of the numbers.
- If both strings do not end with a number, the standard comparison is used.

The following expression uses the comparator above to sort the strings as filenames:

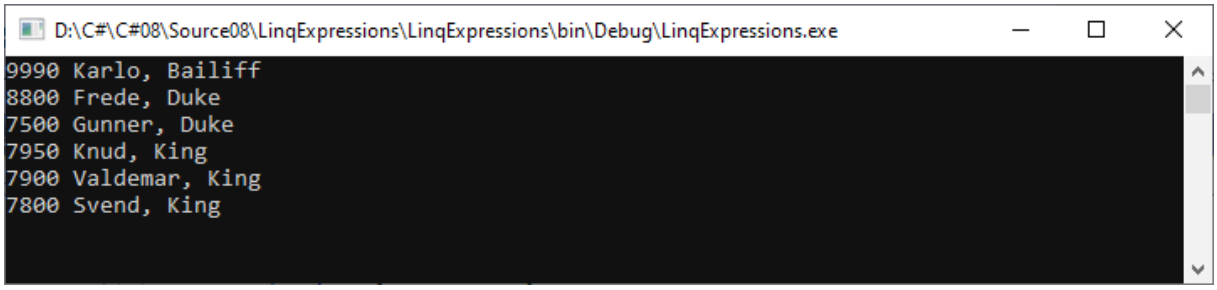
```
var res = names.OrderBy(s => s, new NameOrder());
```



Below is an expression that sorts the result first by *Job* and then by *Zipcode*:

```
var res = from p in pers orderby p.Job, p.Zipcode descending select p;
```

By default, objects are sorted in ascending order, but with *descending* you can specify that they must be sorted in descending order. In this case, the objects are sorted first by *Job* in ascending order and then by *Zipcode* in descending order:



As a final example of an expression regarding sort, the following expression sorts first the persons by *Job* in ascending order, then by *Name* in descending order and finally by *Zipcode* in ascending order:

```
var res =
  pers.OrderBy(p => p.Job).ThenByDescending(p => p.Name).ThenBy(p =>
p.Zipcode);
```

Note in particular the names of the extension methods.

Groups

Above, I have used groups several times, and the following shows several examples of groups - a topic that is actually quite complex with some non-trivial syntax.

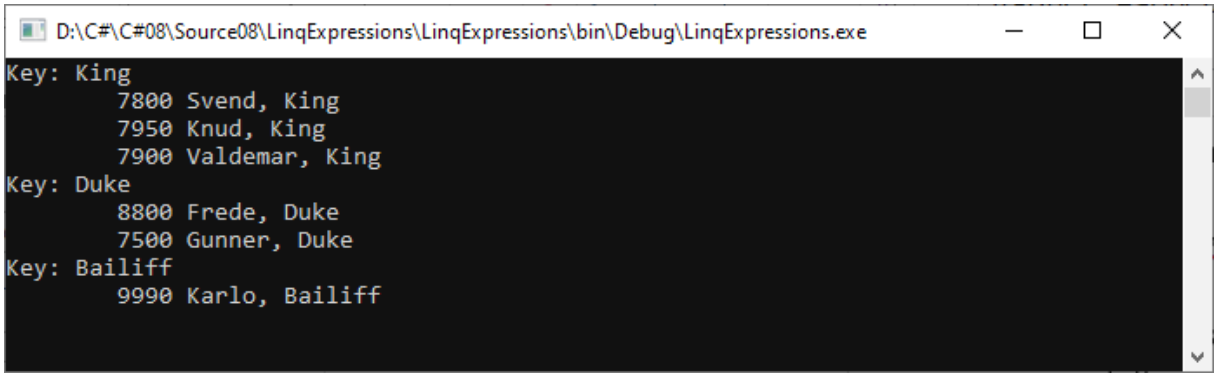
If you group a data source, the result is a collection of the type

IEnumerable<IGrouping<TKey, TElement>>

and thus a collection, where each element is a sub-collection of the data source identified by a key. The following expression

```
static void Test22()
{
  var res = from p in pers group p by p.Job;
  foreach (var g in res)
  {
    Console.WriteLine("Key: " + g.Key);
    foreach (var e in g) Console.WriteLine("\t" + e);
  }
}
```

is an expression that divides the persons into groups where the key is job:



The same expression can be written using an extension method as follows:

```
var res = pers.GroupBy(p => p.Job);
```

You do not have to group by a field, but you can specify an expression. The following query groups the persons by the first letter of the name, so the key is just a character:

```
var res = pers.GroupBy(p => p.Name[0]);
```

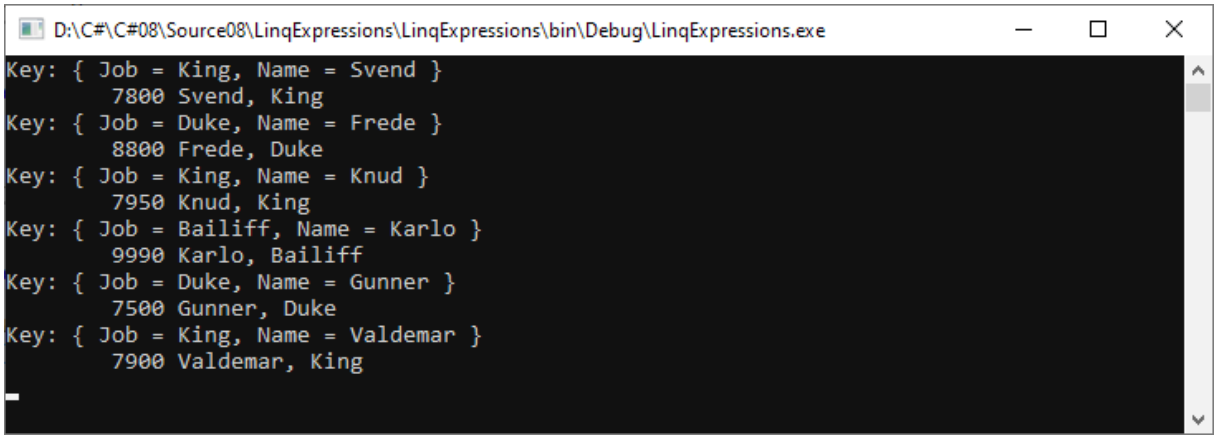
As another example, the following groups by the last three letters of the job title:

```
var res =
  pers.GroupBy(p => p.Job.Length > 2 ? p.Job.Substring(p.Job.Length
- 3) : p.Job);
```

Since the job title does not necessarily contain 3 characters, it is necessary to test for it.

When selecting groups from a data source, the type of key does not have to be a single field. In the following expression, the key is composed and consists of both job and name:

```
var res = from p in pers group p by new { p.Job, p.Name };
```



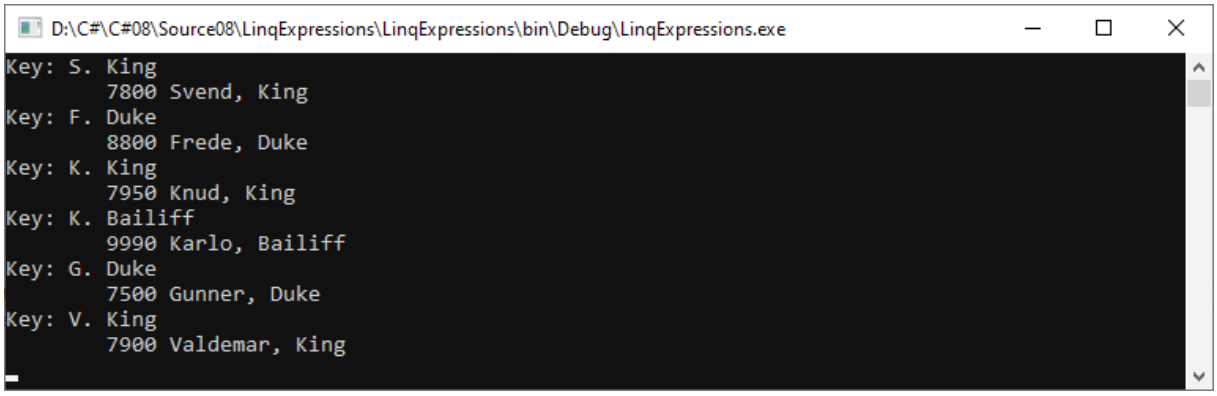
Note that the type of key is an anonymous type. It is also possible to define the key type as a class:

```
class PersonGroup
{
  public string Name { set; get; }
  public string Job { set; get; }

  public override bool Equals(object obj)
  {
    if (!(obj is PersonGroup)) return false;
    PersonGroup group = (PersonGroup)obj;
    if (string.IsNullOrEmpty(Name) && string.IsNullOrEmpty(group.Name))
      return Job.Equals(group.Job);
    if (string.IsNullOrEmpty(Name) || string.IsNullOrEmpty(group.Name))
      return false;
    return Job.Equals(group.Job) && Name[0] == group.Name[0];
  }

  public override int GetHashCode()
  {
    if (string.IsNullOrEmpty(Name)) return Job.GetHashCode();
    return Name[0].GetHashCode() ^ Job.GetHashCode(); ;
  }

  public override string ToString()
  {
    if (string.IsNullOrEmpty(Name)) return Job;
    return string.Format("{0}. {1}", Name[0], Job);
  }
}
```

The class defines a key for grouping *Person* objects, where the key is composed of *Name* and *Job*. Such a key class must override both *Equals()* and *GetHashCode()*, as they are used to determine which objects have the same key. In this case, two *Person* objects must have the same key if they have the same job title and if the names start with the same letter. It is necessary to take special account of how the comparison should be if one of the names is zero or empty, as one can then not compare on the first letter of the name.

The following expression group the persons by this type of key:

```
var res = pers.GroupBy(p => new PersonGroup { Name = p.Name, Job = p.
Job });
```



For the extension method *GroupBy()*, you can also specify a comparator, which specifies how keys are to be compared. A comparator must be a class that implements the interface *IEqualityComparer<T>*, and the following comparator performs a so-called phonetic comparison of strings, where names are compared according to how they sounds:

```
class SoundexComparer : IEqualityComparer<string>
{
  public bool Equals(string s1, string s2)
  {
    return GetHashCode(s1) == GetHashCode(s2);
  }

  public int GetHashCode(string s)
  {
    s = Soundex(s);
    return Convert.ToInt32(s[0]) * 1000 + Convert.ToInt32(s.Substring(1));
  }

  private string Soundex(string s)
  {
    if (string.IsNullOrEmpty(s)) return null;
    StringBuilder builder = new StringBuilder(s.Length);
    s = s.ToUpper().Replace(" ", "");
    if (s.Length == 0) return null;
    char ch = s[0];
    builder.Append(s[0]);
    for (int i = 1; i < s.Length; ++i)
    {
      char n = '0';
      if ("BFPV".Contains(s[i])) n = '1';
      else if ("CGJKQSXZ".Contains(s[i])) n = '2';
      else if ("DT".Contains(s[i])) n = '3';
      else if ("L".Contains(s[i])) n = '4';
      else if ("MN".Contains(s[i])) n = '5';
      else if ("R".Contains(s[i])) n = '6';
```

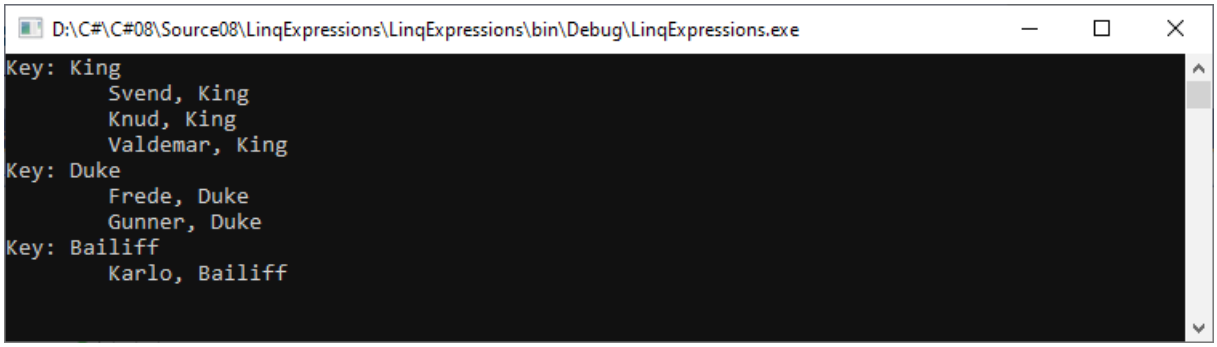
```
      if (n != ch && n != '0')
      {
        builder.Append(n);
        ch = n;
      }
    }
    while (builder.Length < 4) builder.Append('0');
    return builder.ToString(0, 4);
  }
}
```

The names are compared on the basis of an algorithm prepared from an analysis of English names. The next LINQ expression divides an array of names into groups using this comparator:

```
string[] names =
  { "Kirsten", "Kresten", "Knud", "Karlo", "Karl", "Karsten", "Kristian",
    "Karen", "Katrine", "Kalde", "Kristoffer" };
var res = names.GroupBy(s => s, new SoundexComparer());
```

The next expression groups the persons by job, but such individual groups contain objects of the type string. That is the type of groups may be something other than the type of data source.

```
var res = pers.GroupBy(p => p.Job, p => p.Name + ", " + p.Job);
```



Join

As the last basic LINQ operator I will mention join, and it is a very complex operator with a lack of options and correspondingly also an advanced syntax. As the name suggests, there are many similarities with the corresponding database operation, but there are also significant differences.

Basically, the operation is used to combine data from two data sources, but in fact you can use all the data sources that you want if it does not give a performance problem, and there is reason to warn a little against join operations, as efficiency is not always as desired.

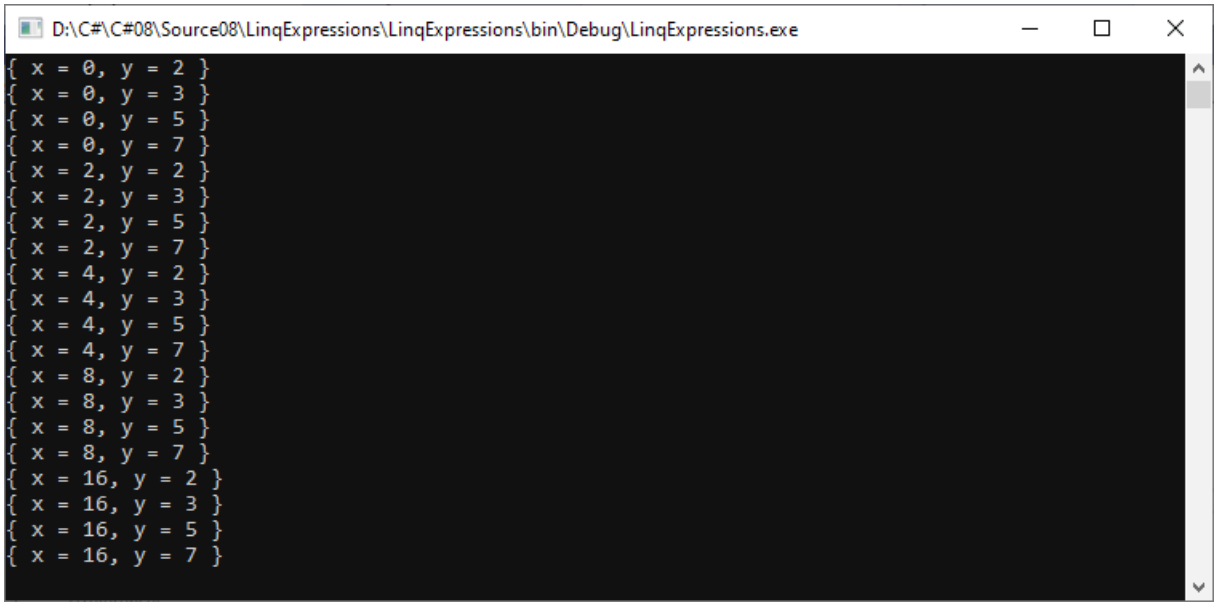
As a starting point, I will look at two data sources, and a join consists, as mentioned, of combining objects in the first data source - the external data source - with objects in the second data source - the internal data source. There are three versions:

1. *Cross Join*, which simply forms the Cartesian product between two data sources - each object in the external data source is combined with each object in the internal data source, and the number of objects in the result is therefore the number of objects in the external data source multiplied by the number of objects in the internal data source.
2. *One-to-one Inner Join,* where objects are combined based on a given common key - each object in the external data source is combined with each object in the internal data source that has the same value for the key.
3. *One-to-many Join*, where a group of objects from the internal data source is assigned from a key value to each object in the external data source and such that the group's key is the key object from the external data source.

A cross join is very simple and you do not actually use the word join. The following expression performs a cross join between two arrays:

```
int[] numbers1 = { 0, 2, 4, 8, 16 };
int[] numbers2 = { 2, 3, 5, 7 };
var res = from x in numbers1 from y in numbers2 select new { x, y };
```

There's not so much mystery in it, and the result is a collection of 20 numbers pairs:

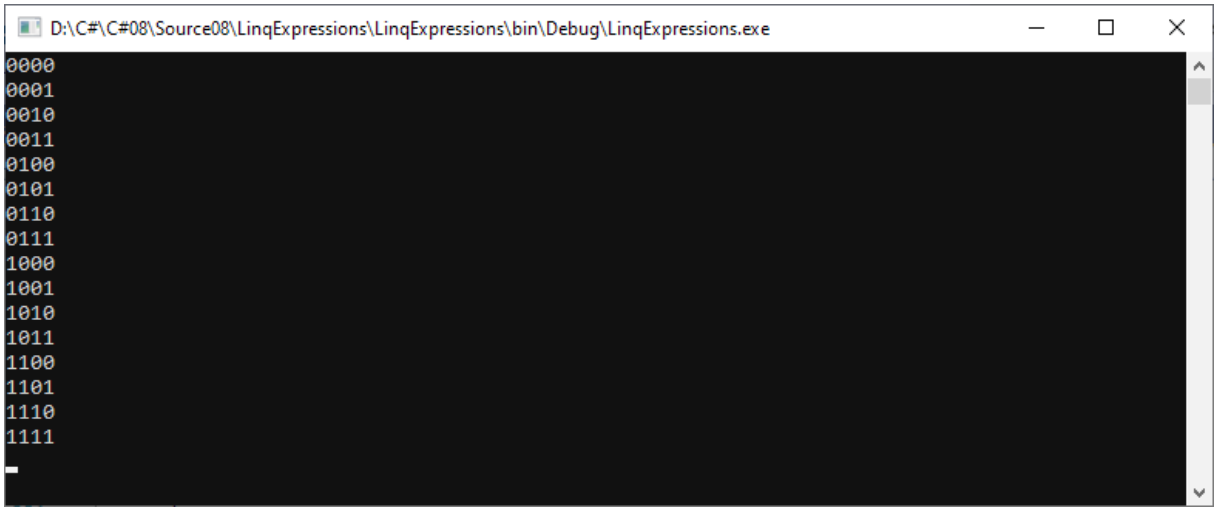


Note that the result consists of a collection of objects of an anonymous type. Incidentally, the same expression can be written in the dot format as follows:

```
var res = numbers2.SelectMany(x => numbers1, (x, y) => new { x, y });
```

As mentioned, all the data sources that one may be interested in can be included, and in the following expression, 4 data sources are included (all of which are the same). The expression therefore determines all possible 4-bit patterns (16 in total):

```
int[] bits = { 0, 1 };
var res = from a in bits from b in bits from c in bits from d in bits
          select string.Format("{0}{1}{2}{3}", a, b, c, d);
```



The most important join operation is one-to-one, which relates data from two data sources to each other via a common key. To illustrate this operation, I need some data sources, and here I will use objects of the type *Person* as well as objects of the following type:

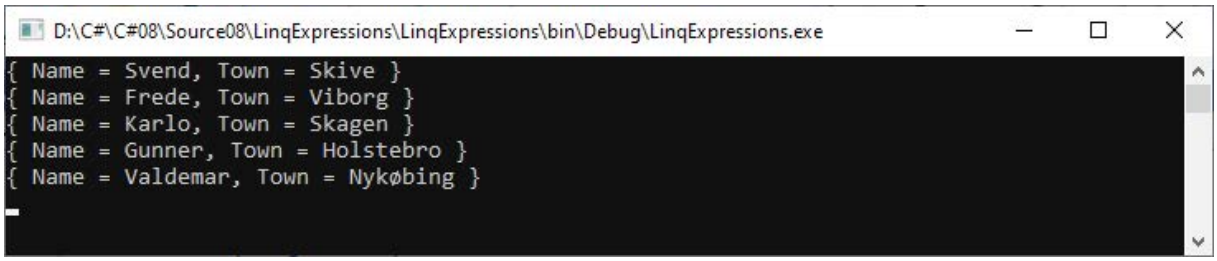
```
class Zipcode
{
  public string Code { get; set; }
  public string Name { get; set; }

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

I will use the data source *pers* as well as the following data source:

```
Zipcode[] post = {
  new Zipcode { Code = "8800", Name = "Viborg" },
  new Zipcode { Code = "9990", Name = "Skagen" },
  new Zipcode { Code = "7900", Name = "Nykøbing" },
  new Zipcode { Code = "7500", Name = "Holstebro" },
  new Zipcode { Code = "7800", Name = "Skive" } };
```

You should first note that there is a person with a zip code that does not appear in the data source *post*. Below is a join of the external data source *pers* with the internal data source *post* and with *zipcode* as key:

```
var res = from p in pers join b in post on p.Zipcode equals b.Code
   select new { Name = p.Name, Town = b.Name };
```



Note that the result has an object for each object in the external data source - except one, as there is no zip code in the data source record with the key 7950. This object is therefore ignored.

The same expression can be written with the dot notation in the following way:

```
var res4 = pers.Join(post, p => p.Zipcode, b => b.Code,
    (p, b) => new { Name = p.Name, Town = b.Name });
```

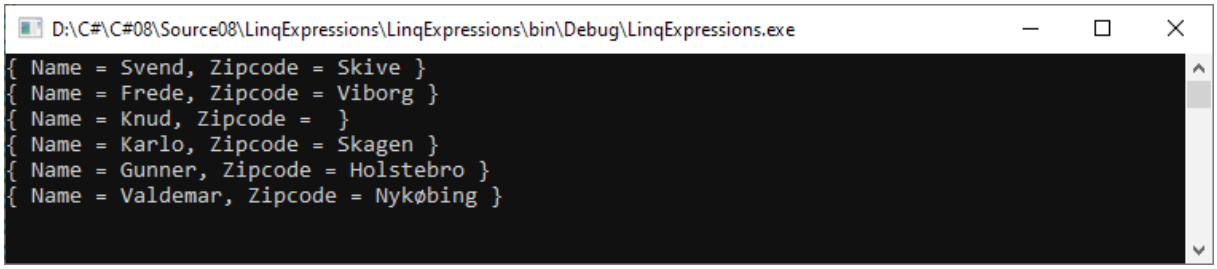
Note that if you switch between the external and internal data source:

```
var res = from b in post join p in pers on b.Code equals p.Zipcode
   select new { Name = p.Name, Town = b.Name };
```

then you get the same result (the same objects), but in a different order - this time each object in the data source *post* will be combined with the objects in the data source *pers*, which have the same zip code.

In database contexts, a (left) external join is a join that includes objects from the external data source, to which no object corresponds in the internal data source. It can also be expressed in a LINQ expression using into and the operator *DefaultIfEmpty()*:

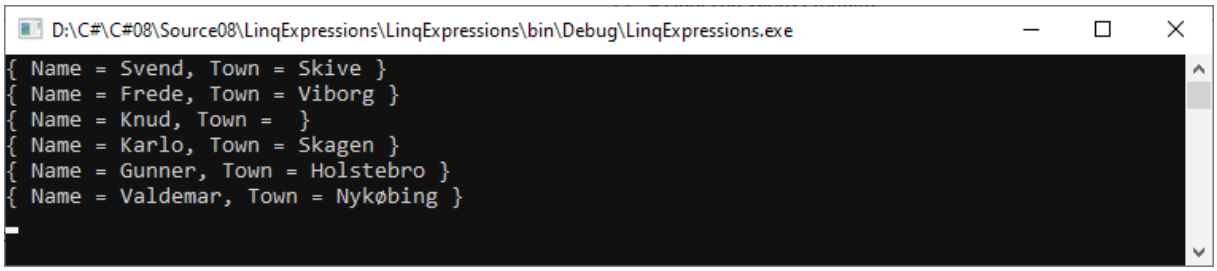
```
var res = from p in pers join b in post on p.Zipcode equals b.Code
into t
    from q in t.DefaultIfEmpty()
    select new { Name = p.Name, Zipcode = q == null ? "" : q.Name };
```



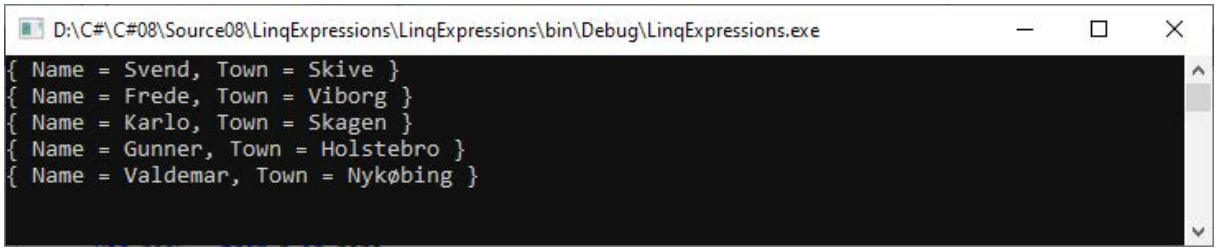
It is also possible to use a composite key a bit in the same way as for groups.

There are other options for writing a one-to-one join:

```
var res = from p in pers select new { Name = p.Name,
     Town = (from b in post where p.Zipcode.Equals(b.Code)
     select b.Name).SingleOrDefault() };
```



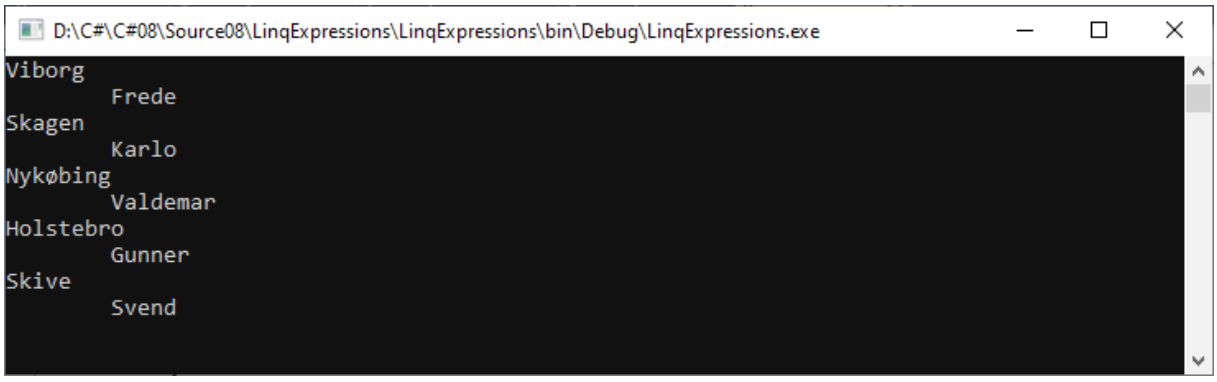

```
var res = from p in pers from b in post where p.Zipcode.Equals(b.Code)
    select new { Name = p.Name, Town = b.Name };
```



where the first expression uses a subquery, while the second is cross join with a *where* part. In particular, you should note that the last expression is easy to read. On the other hand, both of the above expressions have in common that the oversized data sources are extremely inefficient and you should therefore always use the join operator for one-to-one joins.

Finally, I want to show a one-to-many join that for each zip code assigns all persons who have this zip code:

```
var res = from b in post join p in pers on b.Code equals p.Zipcode
into g
    select new { Name = b.Name, Persons = g };

foreach (var b in res)
{
  Console.WriteLine(b.Name);
  foreach (var p in b.Persons) Console.WriteLine("\t" + p.Name);
}
```



This join does not have a very good performance either. It is therefore recommended that you use a so-called lookup table instead:

```
var res = from b in post select new { Name = b.Name, Persons =
lookup[b.Code] };
```

A lookup table is a hash table that in this case contains groups of *Person* objects and with zip code as key. The LINQ expression is now just a plain simple expression that for each zip code retrieves the current group from the lookup table.

EXERCISE 9: QUERY COLLECTIONS

Create a new console application project called *ZipcodesProgram*.

The code examples for this book contains three text files:

1. zipcodes
2. regions
3. municipalities

The files are used and described in the book C# 5 and are copied from there. The first contains the Danish zip codes as a semicolon separated file. The second contains the five Danish regions as a comma separated file (each Danish municipality belongs to to one region) and the last file contains the Danish municipalities as a semicolon separated file. Each line in the last file has at least 5 fields:

1. municipality number
2. municipality name
3. region number
4. area in square kilometers
5. number of inhabitants

These 5 fields can be followed by any number of zip codes which indicate the zip codes used by this municipality.

Copy these three files to the program folder, that is the folder *bin\Debug* under your project folder.

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

```
class Zipcode
{
  public string Code { get; set; }
  public string Name { get; set; }

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

  public override bool Equals(object obj)
  {
    if (obj == null || Code == null) return false;
    if (obj is Zipcode) return Code.Equals(((Zipcode)obj).Code);
    return false;
  }
}
```

Add also a class which represents all zip codes in the file *zipcodes:*

```
class Zipcodes : IEnumerable<Zipcode>
{
  private List<Zipcode> list = new List<Zipcode>();

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

  public IEnumerator<Zipcode> GetEnumerator()
  {
    return list.GetEnumerator();
  }

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

  public void Add(Zipcode zipcode)
  {
    list.Add(zipcode);
  }
}
```

Add corresponding classes for regions and municipalities and also classes (called *Post* and *Posts*) that represent pairs of municipality number and post code. You should then add the following class:

```
class Data
{
  public Zipcodes Zipcodes { get; private set; }
  public Regions Regions { get; private set; }
  public Municipalities Municipalities { get; private set; }
  public Posts Posts { get; private set; }

  public Data()
  {
    Zipcodes = new Zipcodes();
    Regions = new Regions();
    Municipalities = new Municipalities();
    Posts = new Posts();
    Build();
  }

  private void Build()
  {
    // read the three text files and create and fill objects in the above
    // data structures
  }
}
```

In the *Program* class you must create an instance of the class *Data* and you must then write some test methods:

1. Create a query using the query format which extracts all municipalities where the number of inhabitants are greater that 100000 and such the municipalities are sorted by name. For each municipality in the query print the name and the area.
2. Perform the same query as above, but this time using the dot notation.
3. Print the number of inhabitants in Denmark, that is the sum of inhabitants in all municipalities.
4. Print the number of inhabitants in each region (note that you can use *group by*).
5. Create a query using the dot notation such the query extract all municipalities in the region with the number 1081 and such the municipalities are sorted by number of inhabitants and then by area. For each municipality in the query print the name, number of inhabitants and the area.
6. Create the same query, but this time using the query format.

7. Create a query using the query format when the query for all municipalities with region number 1081 should contains the municipality name and the region name.
8. Create the same query, but this time using the dot notation.
9. Create a query using the query format when the query for all municipalities using the zip code for *Skive* should contain the municipality name, the zip code, the number of inhabitants and the area.
10. Create the same query, but this time using the dot notation.

6.3 QUERY OPERATORS

The namespace *System.Linq* defines a very large number of query operators such as *Sum()*, *Max()*, etc. Many of these operators are self-explanatory or at least easy to understand, and I will in no way describe them all, but I will mention a few in this chapter. Partly because they are very useful, and partly because it can be difficult to understand the meaning of them.

The example in this section belonging to the project *LinqOperators*.

Aggregate()

It is an operator that is used to traverse a collection and do something with the individual elements. As an example, the following expression determines the sum of the elements in an array:

```
double[] t = { 2, 3, 5, 7, 11, 13, 17, 19 };
var r0 = t.Aggregate(0.0, (s, x) => s + x);
```

The *Aggregate()* function has two parameters, the first being a start value while the second is a method. The method has two parameters, where the first *s* is the return value from the previous call of the function (where the first object it is the initial value), while the second *x* is the current object. In this case, the method returns the sum of the previous return value and the current value, and the result is that it will return the sum of all the elements.

The result of this expression is thus the same as

```
var r0 = t.Sum();
```

and one can thus see that many of the query operators are simply encapsulations of the *Aggregate()* method.

Above, the aggregation function is indicated as a lambda expression, but one can of course also explicitly refer to a method. Consider the following method:

```
private static double Sqr(double s, double x)
{
  return s + x * x;
}
```

where the return type and parameters all have the same type as the array *t*. The following LINQ expression determines the sum of all squares in the array *t*:

```
var r1 = t.Aggregate(0.0, Sqr);
```

The same expression can be written with a lambda expression:

```
var r2 = t.Aggregate(0.0, (s, x) => s + x * x);
```

It is possible to specify another parameter for the *Aggregate()* method, which refers to a method to be executed on the result at the end. If, for example you have an array as above, you can calculate the standard deviation as follows:

$$\sqrt{\sum_{i=1}^{n}(x_i - m)^2}$$

where m is the mean (average). It is easy to express such a calculation using a LINQ expression:

```
double m = t.Average();
var r3 = t.Aggregate(0.0, (s, n) => { double x = n - m; return s + x
* x; },
                     s => Math.Sqrt(s));
```

The method *Aggregate()* can also work on non-numeric data sources - although this is not the typical application. Below is an example that prints all postcodes in a collection:

```
Zipcode[] post = {  new Zipcode { Code = "8800", Name = "Viborg" },
                    new Zipcode { Code = "7950", Name = "Erslev" },
                    new Zipcode { Code = "7800", Name = "Skive" } };
var r4 = post.Aggregate("", (a, b) => { Console.WriteLine(b); return
null; });
```

Note in particular that the lambda expression returns null - it must return something that is of the type *Zipcode*.

ToArray() and the like

The result of a query is a collection of the type *IEnumerable<T>*, but there are some methods that can convert to other collections. For example the following expression extracts some *Person* objects from a list and returns the result as an array of the type *Person*:

```
List<Person> list = new List<Person> {
  new Person { Zipcode = "7800", Name = "Svend", Job = "Konge" },
  new Person { Zipcode = "8800", Name = "Frede", Job = "Tater" },
  new Person { Zipcode = "7500", Name = "Knud", Job = "Skarpretter" },
  new Person { Zipcode = "8000", Name = "Karlo", Job = "Tater" },
  new Person { Zipcode = "7950", Name = "Gudrun", Job = "Fe" },
  new Person { Zipcode = "8888", Name = "Abelone", Job = "Heks" },
  new Person { Zipcode = "7900", Name = "Valdemar", Job = "Konge" } };
Person[] pers = list.Where(p => p.Zipcode[0] == '7').ToArray();
```

In the same way, an expression is shown below which does the opposite. It extracts some numbers from an array and gives the result in the form of a list:

```
int[] t = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };
List<int> tal = t.Where(n => n > 10).ToList();
```

As the last example, the following expression will take the persons in the list above and create a dictionary of these persons with zip code as key:

```
Dictionary<string, Person> tabel = list.ToDictionary(p => p.Zipcode);
foreach (string key in tabel.Keys) Console.WriteLine(tabel[key]);
```

Here it is important that the values selected as keys are different. Otherwise you get an exception. *ToDictionary()* is somewhat similar to *ToLookup()*, which I mentioned above. The difference is that *ToDictionary()* requires uniqueness of keys, whereas *ToLookup()* associates a group with each key value.

Range and Repeat

I will mention two operators that can be useful to know, e.g. to initialize an array:

```
int[] t1 = Enumerable.Range(2000, 10).ToArray();
int[] t2 = Enumerable.Repeat(3, 5).ToArray();
```

The first statement creates an array of 10 consecutive numbers starting with 2000. The second creates an array of 5 numbers, all of which have the value 3.

Skip and Take

These are two simple operators that can be used to select a specific number of objects from a collection. Below is an expression which, based on a collection consisting of the first 100 positive integers, selects 5 consecutive numbers starting with the number 16:

```
int[] t = Enumerable.Range(1, 100).ToArray();
var r = t.Skip(15).Take(5);
```

You specify how many numbers to skip and then how many to include. Which numbers to skip and which to include can also be specified as a condition, e.g.

```
var s = t.SkipWhile(n => n < 10).TakeWhile(n => n < 20);
```

where one first skips those less than 10 and then takes those less than 20.

Set operators

LINQ also defines a family of set operators that can be used to perform classic set operations on collections with objects of the same type. These operators are not often needed, as .NET implements the type *HashSet<T>*, which represents a set, and if you need a set, you will probably choose this type, but conversely there may also be situations where it may be smart to treat another collection as if it were a set. However, one should be aware of the following:

- A *HashSet<T>* can not contain the same element several times, which is the case with another collections, and if you use an arbitrary collection as a set using the LINQ operators, you must be aware that the same element can occur several times.
- If there is a need to modify the set (add or remove elements), use a *HashSet<T>*.

The conclusion is that although LINQ defines extension methods for set operations, it is a useful alternative to the class *HashSet<T>*, but not a substitute.

The operations are as follows:

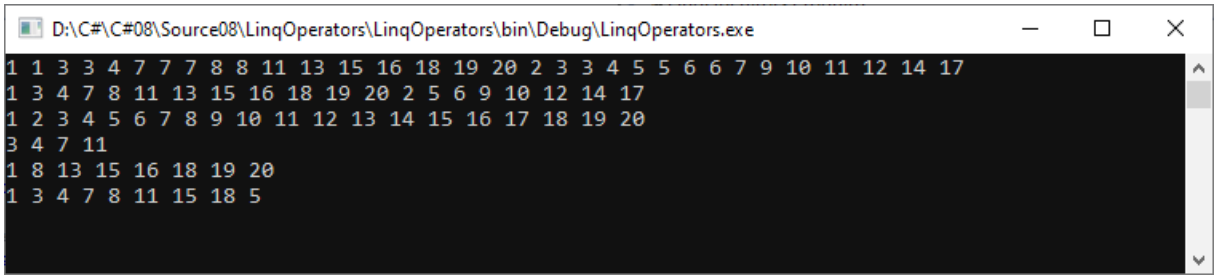
- *Concat()*, which concatenates two sets (puts them in extension of each other), and if the same element occurs in both sets, it will appear several times in the result.
- *Union()*, which is a union and basically works like *Concat()*, but duplicates are removed.
- *Intersect()*, which is intersection of sets and the result only includes elements that occur in both sets, and such that duplicates have been removed.
- *Except()*, which is the set difference and only includes elements that occur in the first set and do not occur in the second set.
- Distinct(), which removes duplicates.

In general, the operators have one parameter, which is a set (the other set), but you can also specify a comparator, which tells you how the comparison should be. The comparator must have the type *EqualityComparer<T>*.

The easiest way is to illustrate the operators with an example:

```
private static void Test5()
{
  int[] t1 = { 1, 1, 3, 3, 4, 7, 7, 7, 8, 8, 11, 13, 15, 16, 18, 19,
  20 };
  int[] t2 = { 2, 3, 3, 4, 5, 5, 6, 6, 7, 9, 10, 11, 12, 14, 17 };
  var q1 = t1.Concat(t2);
  var q2 = t1.Union(t2).OrderBy(t => t);
  var q3 = t1.Intersect(t2);
  var q4 = t1.Except(t2);
  var q5 = t1.Union(t2, new IntegerComparer());
  Print(q1);
  Print(q1.Distinct());
  Print(q2);
  Print(q3);
  Print(q4);
  Print(q5);
}

private static void Print<T>(IEnumerable<T> r)
{
  foreach (T t in r) Console.Write("{0} ", t);
  Console.WriteLine();
}

class IntegerComparer : IEqualityComparer<int>
{
  public bool Equals(int t1, int t2)
  {
    return GetHashCode(t1) == GetHashCode(t2);
  }

  public int GetHashCode(int t)
  {
    return Tvaersum(t);
  }

  private static int Tvaersum(int t)
  {
    int q = t < 0 ? -1 : 1;
    t = Math.Abs(t);
    while (t > 9)
```

```
    {
      int s = 0;
      while (t > 0)
      {
        s += t % 10;
        t /= 10;
      }
      t = s;
    }
    return q * t;
  }
}
```



6.4 USER DEFINED LINQ EXPRESSIONS

LINQ has been designed to be expandable from the start, and it is to be expected that Microsoft will continuously add new expressions as new versions of the language emerge. A LINQ expression is technically just an extension method, which adds functionality to all collections of the type *IEnemerable<T>*, and you can therefore add new LINQ expressions as needed and desired - and it is actually the idea that you should do so, so that you expand with new expressions that reflect and take care of the tasks that you typically work with. If you do this, it is strongly recommended that you follow certain guidelines so that your custom LINQ expression behaves as much as possible as the framework's LINQ expressions. The purpose of this section is to show some examples of custom LINQ expressions.

LINQ statements can generally be divided into four categories determined by the return value:

1. Single element operator that returns a single value.
2. Sequence operator that returns a sequence of elements.
3. Aggregate operator, which returns the result of a data processing (calculation).
4. Grouping operator, which returns a family of groups identified by a key

In the following, I will show a custom operator within each category. The four operators are members of the static class *PaLinq* added to the project *LinqExpressions*.

Single element operator

The first operator is a single element operator, which returns a random element in a collection. It must be a collection that implements the *IEnumerable<T>* interface, and the protocol is

1. if the data source is null, an *ArgumentNullException* is raised
2. if the data source is empty, an *InvalidOperationException* is raised

It is recommended that a single element operator follow this protocol, and in generating test data, it is a useful operator.

```
public static T Random<T>(this IEnumerable<T> data)
{
  if (data == null) throw new ArgumentNullException("Data source is null");
  if (data.Count() == 0) throw new InvalidOperationException("Date
  source is empty");
  return RandomInternal(data);
}
```

RandomInternal() is an auxiliary method that determines a random element in a collection provided it is not empty. For performance reasons, the method is written so that it distinguishes whether it is a collection that implements *IList<T>*. If this is the case, you can reference the item in question with an index. Otherwise, it is necessary to count towards the element.

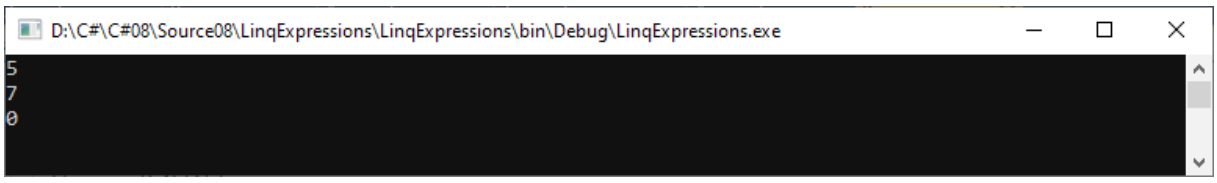
```
private static T RandomInternal<T>(IEnumerable<T> data)
{
  int n = rand.Next(0, data.Count());
  if (data is IList<T>) return ((IList<T>)data)[n];
  using (IEnumerator<T> it = data.GetEnumerator())
  {
    for (it.MoveNext(); n > 0; --n, it.MoveNext()) ;
    return it.Current;
  }
}
```

If you implement a single element operator, it is recommended that you also implement a version which, in the case of an empty data source, returns the default value of the type rather than raising an exception:

```
public static T RandomOrDefault<T>(this IEnumerable<T> data)
{
  if (data == null) throw new ArgumentNullException("Data source is null");
  if (data.Count() == 0) return default(T);
  return RandomInternal(data);
}
```

The following method use the above operators to determine a random number in three collections:

```
static void Test35()
{
  int[] numbers = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  var query = numbers.Select(n => n);
  List<int> list = new List<int>();
  Console.WriteLine(numbers.Random());
  Console.WriteLine(query.Random());
  Console.WriteLine(list.RandomOrDefault());
}
```



Note the last 0 which is the default value for an *int*.

Sequence operator

A sequence operator returns a sequence of the type *IEnumerable<T>* by extracting elements from a data source usually filtered or transformed in some way, and the default operators are *Where()* and *Select()*. These operators use yield return, which allows them to implement iterators, which work in such a way that after returning an element, they suspend the

operation (method) until the next reference (query). This means that it is not necessary to create a new collection object for the result, which can have a great impact on the result. It is recommended that you adhere to this pattern when implementing your own sequence operators.

In this case, the goal is to implement a sequence operator named *SubSequence*, which for a collection works a bit in the same way as the method *Substring()* in the *String* class, that is an operator that returns a subset of objects from a collection. The parameters are a predicate start, which indicates the first element to be included, and a different predicate than which indicates the first element that is not to be included, and if you do not specify a value, the result will contain all objects from the start to the end of the data source. The work itself is performed by an auxiliary method which returns an iterator traversing the data source. The protocol is

1. if the data source is null, an *ArgumentNullException* is raised
2. if the start predicate is null, an *ArgumentNullException* is raised
3. or a yield return is performed

```
public static IEnumerable<T> SubSequence<T>(this IEnumerable<T> data,
                        Predicate<T> start, Predicate<T> end = null)
{
  return end == null ? SequenceIterator<T>(data, start, t => false) :
                        SequenceIterator<T>(data, start, end);
}

private static IEnumerable<T> SequenceIterator<T>(IEnumerable<T> data,
                        Predicate<T> start, Predicate<T> end)
{
  if (data == null)
    throw new ArgumentNullException("Data source is null");
  if (start == null) throw new ArgumentNullException("Start predicate
  is null");
  IEnumerator<T> it = data.GetEnumerator();
  while (it.MoveNext())
  {
    if (start(it.Current))
    {
      yield return it.Current;
      break;
    }
  }
  while (it.MoveNext() && !end(it.Current)) yield return it.Current;
}
```

Below is shown a method which use the method *SubSequence()*:

```
static void Test36()
{
  int[] t = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  var q1 = t.SubSequence(n => n > 9);
  var q2 = t.SubSequence(a => a > 9, b => b > 20);
  var q3 = t.SubSequence(n => n > 30);
  Show(q1);
  Show(q2);
  Show(q3);
}

static void Show<T>(IEnumerable<T> res)
{
  Console.WriteLine("================================");
  foreach (var e in res) Console.WriteLine(e);
  Console.WriteLine("================================");
}
```

Aggregate operator

An aggregate operator performs a data processing on a data source, and in principle it is a matter of traversing the data source and performing that data processing on all elements. Often - and this also applies to the standard aggregate operators - such an operator acts on collections where the elements are numbers and the operator performs some form of calculation. This can make the implementation extensive, as it is often necessary / desirable to implement overrides for all number types. As an example, I will implement an operator that returns the standard deviation of a distribution, which (as mentioned above) is calculated as follows:

$$\sqrt{\sum_{i=1}^{n}(x_i - m)^2}$$

where $x_1, x_2, ..., x_n$ are observations from the distribution, and m is the mean (average).

An aggregate operator returns a value that in this case is the standard deviation of the numbers in a collection. Since the operator works on collections with numeric objects, it should usually be overridden for number types to both integers and decimal numbers. In

this case, I have only implemented the operator for *double*. An aggregate operator will often have a parameter, which is a selector (a predicate) and is used to select which numbers to include in the calculation. In this case, it has the default value of *null*, which indicates that all numbers should be included in the calculation. Otherwise the protocol is as above:

1. if the data source is null, an *ArgumentNullException* is raised
2. if the data source is empty, an *InvalidOperationException* is raised

The operator can then be written as follows:

```
public static double StandardDeviation(this IEnumerable<double> data,
  Predicate<double> select = null)
{
  return Deviation(DoubleIterator(data), select);
}

private static IEnumerable<double> DoubleIterator<T>(IEnumerable<T>
data)
{
  if (data == null) throw new ArgumentNullException("Data source is null");
  if (data.Count() == 0)
    throw new InvalidOperationException("Data source is empty");
  IEnumerator<T> it = data.GetEnumerator();
  while (it.MoveNext()) yield return Convert.ToDouble(it.Current);
}

private static double Deviation(IEnumerable<double> numbers,
  Predicate<double> select)
{
  double my = numbers.Average();
  double sum = 0;
  foreach (double x in numbers)
    if (select == null || select(x)) sum += Sqr(x - my);
  return System.Math.Sqrt(sum / numbers.Count());
}

private static double Sqr(double x)
{
  return x * x;
}
```

There is not much to explain, but both overrides use an iterator to traverse the current collection.

The following method uses the LINQ operator *StandardDeviation* to calculate the standard deviation of a *double* array:

```
static void Test37()
{
  double[] t = { 2.67, 2.91, 3.23, 3.81, 3.99, 2.67, 2.01, 2.45,
  3.91, 3.35 };
  Console.WriteLine(t.StandardDeviation());
}
```

Grouping operator

This family of operators divides the objects in a sequence into groups based on some criterion. The basic operators are *GroupBy()* and *ToLookup()*, and in fact there is rarely a need to define ones own operators. The following should therefore be seen more as an example of how to do.

A grouping operator returns a family of groups represented as a

IEnumerable <IGrouping<K, T >>

sequence. The first step is therefore to write a type that can represent the groups, and the second step is to write an enumeration that divides the objects into groups and returns these groups with a yield return statement.

As an example, I would look at an operator that divides the objects of the data source into groups based on an array of predicates. Each predicate defines a group such that the objects of the group are the objects in the sequence that satisfy the predicate. Each group has a key, which is the index of the defining predicate, however, such that the group with key 0 is always a default group for the objects that do not satisfy a predicate. This grouping is not necessarily unambiguous, corresponding to the fact that there may be objects that fulfill several predicates, and if this is the case, the object is placed in the group for the first predicate (the predicate with the lowest index).

The following class defines a group for a LINQ grouping operator, and the class can generally be used to define grouping operators. The class implements the interface *IGrouping*, which defines the basic group operations. The class also implements the interface *IList* so that one can traverse a group as a list. Since an *IList* defines several properties that are not desirable in this context, they are defined as *unsupported*.

```
public class Grouping<K, T> : IGrouping<K, T>, IList<T>
{
  private K key;
  private List<T> elements = new List<T>();

  public Grouping(K key)
  {
    this.key = key;
  }

  internal void Add(T element)
  {
    elements.Add(element);
  }

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

  public IEnumerator<T> GetEnumerator()
  {
    foreach (T element in elements) yield return element;
  }

  K IGrouping<K, T>.Key
  {
    get { return key; }
  }

  int ICollection<T>.Count
  {
    get { return elements.Count; }
  }

  bool ICollection<T>.IsReadOnly
  {
    get { return true; }
  }
```

```
bool ICollection<T>.Contains(T elem)
{
  return elements.Contains(elem);
}

void ICollection<T>.CopyTo(T[] array, int index)
{
  Array.Copy(elements.ToArray(), 0, array, index, elements.Count);
}

int IList<T>.IndexOf(T elem)
{
  return elements.IndexOf(elem);
}

T IList<T>.this[int n]
{
  get
  {
    if (n < 0 || n >= elements.Count)
      throw new ArgumentOutOfRangeException("Grouping.this[]");
    return elements[n];
  }
  set
  {
    throw new NotSupportedException();
  }
}

void ICollection<T>.Clear()
{
  throw new NotSupportedException();
}

void ICollection<T>.Add(T elem)
{
  throw new NotSupportedException();
}

bool ICollection<T>.Remove(T elem)
{
  throw new NotSupportedException();
}
```

```
  void IList<T>.Insert(int n, T item)
  {
    throw new NotSupportedException();
  }

  void IList<T>.RemoveAt(int n)
  {
    throw new NotSupportedException();
  }
}
```

Note that internally, a group is represented by a key and a list of the group's items. Note in particular the method *Add()*, which is defined *internal*, as it should not be usable from the client code. Its visibility is thus limited to the assembly which contains the class.

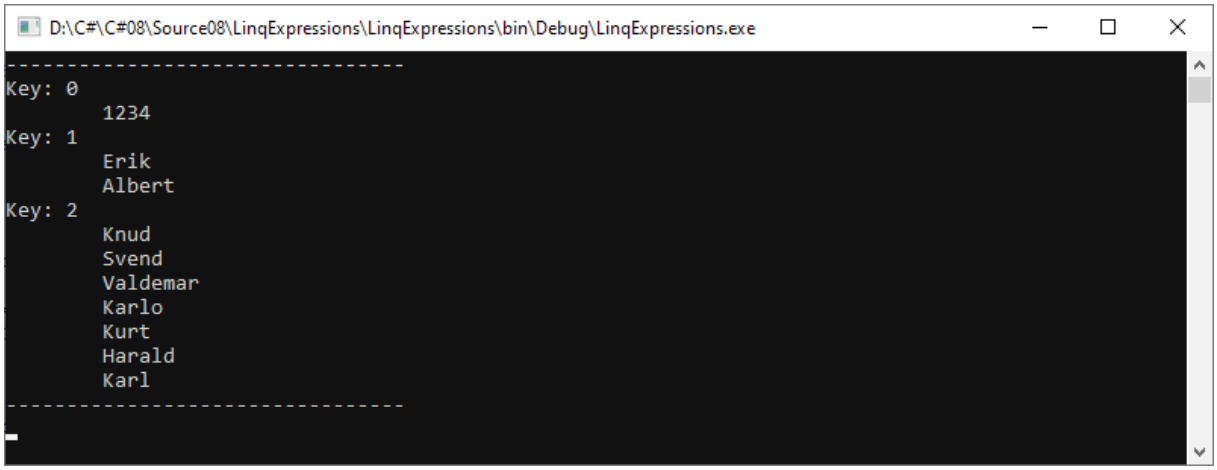
Back there is the method (and its auxiliary method) that is the operator itself. Note in particular the parameter *Select*, which is an array of predicates. There is a group for each predicate, as well as a default group which contains all objects that are not fulfilled a predicate. The keys are 0 (default), 1, 2, 3, which correspond to the predicate's index in the parameter array.

```
public static IEnumerable<IGrouping<int, T>> Category<T>(this
IEnumerable<T> data,
                              params Predicate<T>[] Select)
{
  if (data == null) throw new ArgumentNullException("Data source is null");
  return CategoryIterator(data, Select);
}

private static IEnumerable<IGrouping<int, T>> CategoryIterator<T>
    (IEnumerable<T> data, Predicate<T>[] Select)
{
  Grouping<int, T>[] groups = new Grouping<int, T>[Select.Length + 1];
  for (int i = 0; i < groups.Length; ++i) groups[i] = new
  Grouping<int, T>(i);
  using (IEnumerator<T> it = data.GetEnumerator())
    while (it.MoveNext()) groups[GroupIndex(it.Current, Select)].
    Add(it.Current);
  for (int i = 0; i < groups.Length; ++i) yield return groups[i];
}
```

```
private static int GroupIndex<T>(T element, Predicate<T>[] Select)
{
  for (int i = 0; i < Select.Length; ++i) if (Select[i](element))
  return i + 1;
  return 0;
}
```

The following code divides some names into two groups, one group consisting of all names beginning with a vocal, while the other group consisting of all names beginning with a consonant.

```
static void Test38()
{
  string[] names = { "Knud", "Svend", "Valdemar", "Karlo", "Erik",
  "Albert", "Kurt", "Harald", "Karl", "1234" };
  var groups = names.Category(Vocals, Consonants);
  Show(groups);
}

private static bool Vocals(string navn)
{
  char[] vocals = { 'a', 'e', 'i', 'o', 'u', 'y', 'æ', 'ø', 'å' };
  return vocals.Contains(char.ToLower(navn[0]));
}

private static bool Consonants(string navn)
{
  char[] consonants = { 'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l',
  'm', 'n', 'p', 'q', 'r', 's', 't', 'v', 'w', 'x', 'z' };
  return consonants.Contains(char.ToLower(navn[0]));
}

private static void Show<TKey, TElement>(IEnumerable<IGrouping<TKey,
TElement>> res)
{
  Console.WriteLine("---------------------------------");
  foreach (var g in res)
  {
    Console.WriteLine("Key: " + g.Key);
    foreach (var e in g) Console.WriteLine("\t" + e);
  }
  Console.WriteLine("---------------------------------");
}
```



EXERCISE 10: MORE LINQ EXPRESSIONS

Create a copy of the project from exercise 9. Add test methods for the following queries:

1. A query which prints the municipality number and the number of zip codes used by this municipality, but only for the municipality with the highest number of zip codes and only for zip codes greater than 2000.
2. Add the class *PaLinq* from above to the project. Write a query that prints the average area for all municipalities as well as the standard deviation.
3. A query which for all municipalities extracts the name, area, inhabitants and the population density per square kilometer.
4. A query which extracts all zip codes where the code start with 7 when you should write the query using *SkipWhile()* and *TakeWhile()*.

6.5 THE LAST THING

As shown above, LINQ is a highly efficient extension of C#, especially combined with databases and the Entity Framework. Another question is how effective LINQ is in terms of performance, and LINQ is extremely efficient, but you should still need to think about it, and in particular you should pay attention to join operations.

There are many lines of code behind the implementation of LINQ and although the code is effective, it obviously costs. Below I have shown an example which creates two arrays of each 50000000 elements which are initialized with the numbers 1 - 50000000. For one array it happens with a usual *for* loop while with the other it happens with a LINQ expression. Next the program performs three loops over the last array which performs a calculation on each element. The first uses a regular *for* loop, the next a *foreach* and the last a LINQ expression.

```
namespace LinqPerformance
{
  class Program
  {
    static void Main(string[] args)
    {
      Stopwatch sw = new Stopwatch();
      sw.Start();
      int[] a = new int[50000000];
      for (int i = 0, j = 1; i < a.Length; ++i, ++j) a[i] = j;
      sw.Stop();
      Console.WriteLine(sw.ElapsedMilliseconds);
      sw.Restart();
      int[] arr = Enumerable.Range(1, 50000000).ToArray();
      sw.Stop();
      Console.WriteLine(sw.ElapsedMilliseconds);
      double sum1 = 0;
      sw.Restart();
      for (int i = 0; i < arr.Length; ++i) sum1 += Math.Sqrt(arr[i]);
      sw.Stop();
      Console.WriteLine(sw.ElapsedMilliseconds + "    " + sum1);
      double sum2 = 0;
      sw.Restart();
      foreach (int t in arr) sum2 += Math.Sqrt(t);
      sw.Stop();
      Console.WriteLine(sw.ElapsedMilliseconds + "    " + sum2);
      double sum3 = 0;
      sw.Restart();
      sum3 = arr.Aggregate(0.0, (s, x) => s + Math.Sqrt(x));
      sw.Stop();
      Console.WriteLine(sw.ElapsedMilliseconds + "    " + sum3);
      Console.ReadLine();
    }
  }
}
```

```
D:\C#\C#08\Source08\LinqPerformance\LinqPerformance\bin\Debug\LinqPerformance.exe
185
557
227   235702263930,837
230   235702263930,837
436   235702263930,837
```

You should note that *for* and *foreach* have the same performance while LINQ costs a bit.

7 PARALLEL PROGRAMMING

As mentioned, a running program is usually referred to as a process, and a process has associated an address space which contains the program code, the data on which the program works and various control structures used by the operating system to administer the system's processes. If you take a snapshot of a machine, there will usually be many processes, as there will always be many running programs in the form of user programs, services and the operating system's own programs. There will always be more processes than there are processors, and the processes will therefore in turn be assigned a CPU, something which is controlled by the operating system. Modern operating systems are preemptive operating systems, which means that a process is interrupted if it has had a CPU for a certain period of time, and another process then becomes running instead. This is to ensure that all processes are periodically running, so that for us as users it looks like the processes run parallel next to each other. This is pseudo-parallelism, because although machines can have several CPUs and therefore can run several processes simultaneously, the number of processes is always far greater than the number of CPUs. A process can also be interrupted - deprived its CPU - for reasons other than it has used the slice of its time, e.g. if the process has to perform an IO operation, which is a slow operation relative to CPU instructions, and in such a situation, the process is blocked and another gets the CPU until the IO operation is performed. As mentioned, it is the task of the operating system to manage all that and including also to decide which process to start when a process is interrupted. It is referred to as process scheduling.

A Windows process can have multiple threads and will as mentioned have at least one called the primary thread, where a thread is a code executed as a standalone module by the operating system. Similarly, each thread has its own stack. In Windows, it is threads that are scheduled, and thus it is for a thread that is allocated time on a CPU. Since modern computers have processors with several cores, this in practice allows for true simultaneity, as a process can in principle simultaneously run several threads each on their core. It is basically the job of the operating system to schedule when a thread is running and on what core it is running, but it may be the programmer's job to synchronize the activities of threads to the extent that they use shared resources.

Parallelism can therefore mean a challenge for the programmer, but conversely, parallelism is exactly the goal of threads. Process address spaces are generally protected relative to each other, so that a process cannot address into the address space of another process, but if a process has multiple threads, these threads all run within the same address space and thus have access to all resources within this address space. This is what you want to achieve, and at the same time it is also what can cause problems and make demands for synchronization

of the threads' use of common resources. Another reason for threads is that for the operating system it is cheaper to create a thread than to create a process, so threading can mean improved performance.

When a running thread is interrupted (interrupted by the operating system) for one reason or another, its state in the form of registers and other data structures must be saved so that it can be restarted at a later time, and correspondingly registers must be loaded with content from a previous interrupted thread. This switching of a thread from running to ready (or another mode such as blocked) and restarting a waiting thread is called in Windows a context switch. It is a relatively expensive operation as it involves the kernel and thus also a switch to kernel mode. In later versions of Windows this has led to the introduction of more efficient user mode thread scheduling. Seen from the outside, threads have thus become more difficult to see through, and the whole thing is complicated by the fact that today we often program towards a virtual machine such as CLR.

Processes that execute managed code are no different in terms of threads than usual Windows processes, and CLR will create several threads and e.g. for garbage collection. Threads that run managed code are usually called managed threads, while other threads are called native threads. In fact, a managed thread is a native thread that is extended with additional properties and facilities for the sake of CLR. This means for example that more scheduling activities can be performed by CLR and thus again help to increase the program's performance. In most cases, therefore, it is not important as a programmer to distinguish between managed and native threads, and perhaps the most important difference is that managed threads help to make life easier for the programmer.

As mentioned, it is relatively expensive to create a thread, as it requires the creation of a stack for the thread as well as other data structures and in that context a switch to kernel mode, just as the operating system must include the new thread in the scheduling. To remedy this and in some contexts to increase efficiency, .NET uses a thread pool that can be thought of as a family of pre-allocated threads. In this way, a large part of the work concerning threading is left to the runtime system, as it now is responsible for finding a free thread in the pool.

Using a thread pool is not a solution to all problems, and it is a technique that is particularly suitable in the case of threads that terminate quickly. In the case of threads that run for a long time, the overhead by letting the operating system create a thread is smaller. The thread pool also has a disadvantage as it limits the parallelism to some extent as the runtime system will only attach a limited number of threads to physical threads at a time.

Today it is possible to use true parallelism where several threads run in parallel on each their CPU. The following is a very short and very general introduction to these concepts, which through a few examples describes what it is about, and it is important to be aware that the material requires knowledge of the basic concepts of threads and concurrency control. The type of threads and synchronization described above is necessary, but not sufficient, as it does not necessarily mean increased efficiency, even if the machine has several cores. The solution is

- to divide the code into partitions
- execute these partitions in parallel as threads on multiple cores
- combines results in an efficient way as the individual threads are completed

Although in principle it is possible with the classic mechanisms for threads, it is not easy and this is where PFX comes into play which is a .NET framework. In this context, we talk about two concepts

1. data parallelism, which in short means that if a task works on a large amount of data, the work can be parallelized, where several threads perform the same thing, but each works on a subset of data
2. task parallelism, where the task is divided into several parts, and where each part of the task is performed in its own thread

Here, data parallelism is the easiest to implement, while task parallelism requires more thoughtfulness and synchronization. The following alone introduces data parallelism.

PFX basically consists of two layers, where the top layer is an API called *PLINQ* as well as a class *Parallel*. The bottom layer contains classes for task parallelism as well as some additional help classes. So I will only be interested in the top layer.

PFX has several applications, including the same as illustrated in connection with traditional threading of programs, but the most important is parallel programming, where you have true simultaneity when using several CPU cores, so that several parts of the program can be executed simultaneously. It complicates the programming, and it is these challenges that PFX must, if not solve, then help make it possible. In order to achieve increased performance by means of parallel programming, it is necessary that the algorithm be parallelizable and can be divided into parts which can be performed independently of each other on their respective cores.

7.1 PLINQ

PLINQ is a technique that automatically parallelizes LINQ queries, and it is therefore a technique that is easy to use, as the framework takes care of everything as partitioning, threading and combining the result. All that is needed is to call *AsParallel()* on the input sequence and then the LINQ expression can be written as usual.

As an example, I will show a program called *PlinqPrimes* that, with a LINQ expression, determines all prime numbers that are less than 20 million. The program determines the prime numbers in two ways: Partly with a usual LINQ expression, and partly with a PLINQ expression:

```
class Program
{
  static void Main(string[] args)
  {
    IEnumerable<int> numbers = Enumerable.Range(1, 20000000);
    Test1(numbers);
    Test2(numbers);
  }

  static void Test1(IEnumerable<int> numbers)
  {
    Stopwatch sw = new Stopwatch();
    sw.Start();
    var primes = from n in numbers where IsPrime((ulong)n) select n;
    sw.Stop();
    Console.WriteLine("{0} primes in {1} milliseconds",
       primes.Count(), sw.Elapsed.TotalMilliseconds);
  }

  static void Test2(IEnumerable<int> numbers)
  {
    Stopwatch sw = new Stopwatch();
    sw.Start();
    var primes = from n in numbers.AsParallel() where IsPrime((ulong)
    n) select n;
    sw.Stop();
    Console.WriteLine("{0} primes in {1} milliseconds",
       primes.Count(), sw.Elapsed.TotalMilliseconds);
  }
```

```
  static bool IsPrime(ulong n)
  {
    if (n == 2 || n == 3 || n == 5 || n == 7) return true;
    if (n < 11 || n % 2 == 0) return false;
    for (ulong k = 3, m = (ulong)Math.Sqrt(n) + 1; k <= m; k += 2)
      if (n % k == 0) return false;
    return true;
  }
}
```

Main() creates a collection with all positive integers less than 20000000. Next, the two test methods are called with this collection as a parameter. The test methods measure how long it takes to perform the LINQ expression and then print the result in terms of the number of prime numbers found and the number of milliseconds spent on the expression.

The two methods are largely the same, and the only difference is that *Test2()* uses the method *AsParallel()* on the input sequence *numbers*. This means that the test is performed in parallel and, if possible, uses several cores. If you run the program, the last method is the fastest, but there is not much difference in time, but the last one is a bit faster. When the difference is not greater, it is because it is limited to what extent the task can be parallelized.

In fact, *AsParallel()* is an extension method, which means that the usual LINQ operators are replaced by special parallel programming operators. *AsParallel()* parallelizes a LINQ expression completely transparent. When it is not default, it is because parallel LINQ expressions only make sense if there are enough calculations performed by independent threads, and for many LINQ queries will the overhead by partitioning and then aggregating the result exceeds the gain of multiple parallel threads. If you execute a LINQ query in parallel, you do not necessarily get the same scheme of the result as usual. You can achieve this by e.g. writing

```
numbers.AsParallel().AsOrdered()
```

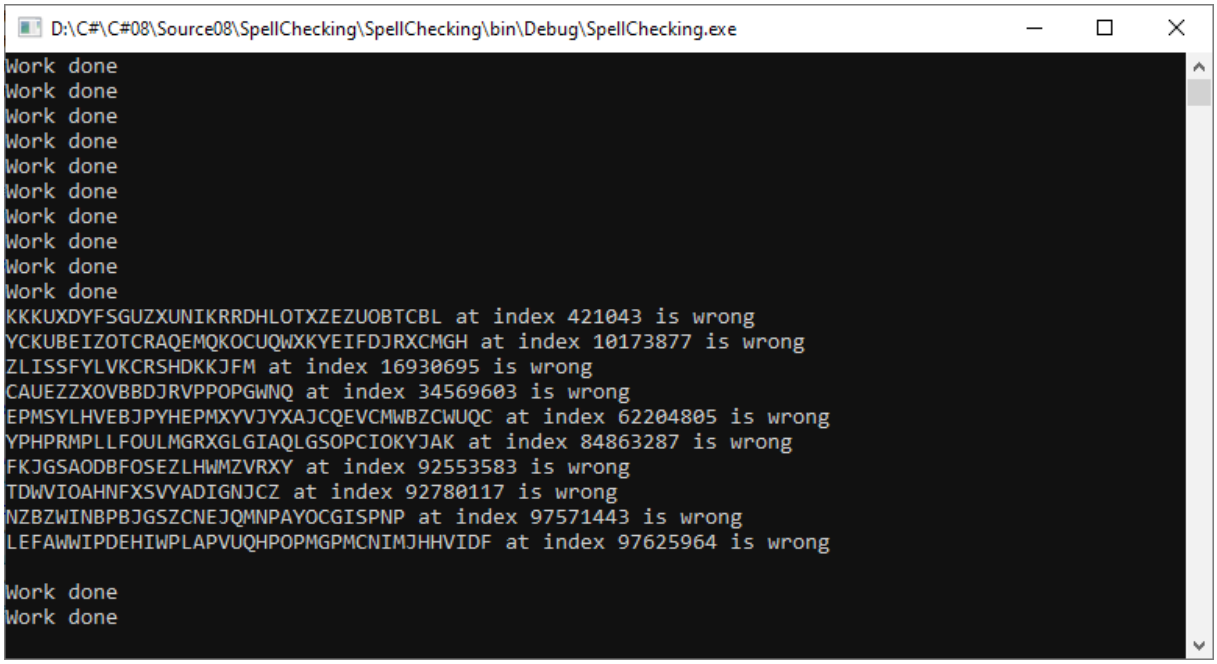
but it obviously affects performance, so you should only use *AsOrdered()* if needed.

7.2 SPELL CHECK

The following example shows how to simulate a program that performs a spell check using data parallelism. The example should show how it happens while the program is doing other work.

```
namespace SpellChecking
{
  class Program
  {
    static Random rand = new Random();
    const int M = 100000;       // number of words in the dictionary
    const int N = 100000000;    // number of words in the document
    const int K = 10;           // number of words misspelled

    static void Main(string[] args)
    {
      (new Thread(new ThreadStart(Worker))).Start();
      HashSet<string> words = CreateList(M);
      string[] doc = CreateDoc(words, N);
      WrongWords(doc, K);
      IEnumerable<DocWord> res = Spelling(doc, words);
      Print(res);
    }

    static void Worker()
    {
      while (true)
      {
        for (int i = 0; i < 10000000; ++i) Math.Cos(Math.Log(Math.Sqrt(i)));
        Console.WriteLine("Work done");
      }
    }

    static IEnumerable<DocWord> Spelling(string[] doc, HashSet<string>
    words)
    {
      return doc.AsParallel().Select(
        (word, index) => new DocWord { Word = word, Index = index }).Where(
          w => !words.Contains(w.Word)).OrderBy(w => w.Index);
    }

    static void Print(IEnumerable<DocWord> words)
    {
      foreach (var w in words)
        Console.WriteLine("{0} at index {1} is wrong", w.Word, w. Index);
      Console.WriteLine();
    }
```

```
    static void WrongWords(string[] doc, int n)
    {
      for (int i = 0; i < n; ++i) doc[rand.Next(0, doc.Length)] =
      CreateWord();
    }

    static string[] CreateDoc(HashSet<string> words, int n)
    {
      string[] arr = words.ToArray();
      var local = new ThreadLocal<Random>(
        () => new Random(Guid.NewGuid().GetHashCode()));
      return Enumerable.Range(0, n).AsParallel().Select(
        i => arr[local.Value.Next(0, arr.Length)]).ToArray();
    }

    static HashSet<string> CreateList(int n)
    {
      HashSet<string> set = new HashSet<string>();
      for (int i = 0; i < n; ++i) set.Add(CreateWord());
      return set;
    }

    static string CreateWord()
    {
      StringBuilder builder = new StringBuilder(rand.Next(20, 40));
      while (builder.Length < builder.Capacity) builder.
      Append(CreateChar());
      return builder.ToString();
    }

    static char CreateChar()
    {
      return (char)rand.Next('A', 'Z' + 1);
    }
  }

  struct DocWord
  {
    public string Word { get; set; }
    public int Index { get; set; }
  }
}
```



There are many details this time.

In *Main()*, a thread is first started that periodically writes the text *Work done* on the screen. In addition, the thread performs a very large number of calculations (which take time), and the goal is to simulate that the program is doing something in parallel with the spell check being performed.

At the end of the program there is a method *CreateChar()* which returns a random letter from A to Z. This method is used in *CreateWord()* to create a word which consists of 20 - 40 random letters, and *CreateWord()* is used in the method *CreateList()* to create a *HashSet<string>* consisting of 100000 words that represents a dictionary.

After the program in *Main()* has created the dictionary, *CreateDoc()* is called to create the document, which is an array of 10000000 words from the dictionary. The document is created with PLINQ as a parallel query, but I will return to that in a moment. After the document is created, the method *WrongWords()* is called, which changes 10 words in the document, and thus has to simulate that there are 10 words that are misspelled.

Next, the spelling check is performed in the method *Spelling()*, which returns the misspelled words, after which they are printed on the screen. The starting point for *Spelling()* is thus a document with 10 million words, and it is the task of the method to return a collection with the words that are not in the dictionary. This must necessarily be done by going through the document and examining for each word whether it appears in the dictionary. Lookup in the dictionary is extremely effective as the dictionary is a *HashSet*. Starting the

spell checker in parallel with the program and so that the checker utilizes all CPU cores in the best possible way is very simple with PLINQ and *AsParallel()*. The result of the query is objects of type *DocWord*, which is a struct with two fields, which represent resp. a word and the position of the word in the document. If you look at the LINQ query itself:

```
doc.AsParallel().Select((word, index) => new DocWord { Word = word,
Index = index }).
  Where(w => !words.Contains(w.Word)).OrderBy(w => w.Index);
```

then it says that the query should be executed in parallel on all CPUs, and that one should select objects of the type *DocWord* for the words that are not found in the dictionary, and the result should eventually be sorted by the index. You should note that such a query is exactly apt to parallelize, since checking whether a word is in the dictionary is independent of checking other words.

Strictly speaking, the type *DocWord* is not necessary, as the compiler would instead create an anonymous type, but it would instead be a class type allocated on the heap rather than the stack allocated type, which in principle would affect the efficiency of the program. At a usual sequential LINQ query, the difference would hardly matter more, but at parallel LINQ the gain is greater corresponding to that stack allocation of objects in itself is parallel, since each thread has its own stack.

The creation of the dictionary is also parallelized. If it was written sequentially, it could be written as:

```
Enumerable.Range(0, n).Select(i => arr[rand.Next(0, arr.Length)]).
ToArray();
```

which in principle is not so much new in. If you want to parallelize this expression, it would be natural to try:

```
Enumerable.Range(0, n).AsParallel().
Select(i => arr[rand.Next(0, arr.Length)]).ToArray();
```

but it will not work as intended, as the method *rand.Next()* is not thread safe. One solution is:

```
var local = new ThreadLocal<Random>(() => new Random(Guid.NewGuid().
GetHashCode()));
```

which creates a *Random* object for each thread. Note the argument to the constructor of the class *Random*, which ensures that two *Random* objects created at the same time initialize different sequences of random numbers. The query can then be written as a parallel query:

```
Enumerable.Range(0, n).AsParallel().
Select(i => arr[local.Value.Next(0, arr.Length)]).ToArray();
```

EXERCISE 11: THE EFFECT OF PLINQ

Create a copy of the project *SpellChecking*. You should change the program to measure the time used to check the document. It is not quite simple as you have to be careful about where you measure the time, as PLINQ is of course based on threads.

Add a variable of the type *StopWatch* to the program. In *Main()* you must start the stopwatch just before you start the spell checking. The method *Print()* has a loop where to print the misspelled words. In the first iteration of this loop you must stop the stopwatch and print the number of milliseconds for the check.

Run the program and test how many milliseconds the program has used the spell checker.

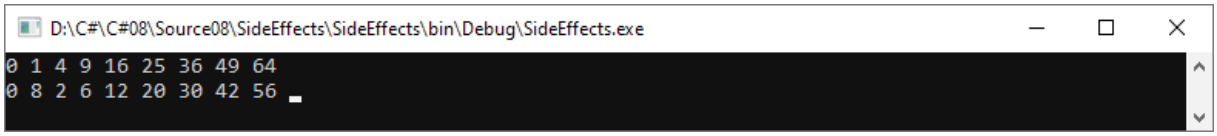
Change the method *Spelling()* such the query no longer use *AsParallel()*. Run the program again. Can you observe any difference?

7.3 THE USE OF PLINQ

It is not so easy to determine when it is advantageous to use PLINQ over usual sequential LINQ queries, primarily because most sequential LINQ queries are already highly efficient. To benefit from PLINQ, look for CPU-tied bottlenecks and try to parallel them. PLINQ

is therefore not always a good solution. Since PLINQ executes a query as parallel threads, one must be especially careful with operations that are not thread safe, and e.g. updating a variable can have unintended side effects. Consider the following program that executes two queries, where each query determines the first 9 square numbers:

```
static void Main(string[] args)
{
  int i = 0;
  int j = 0;
  var q1 = from n in Enumerable.Range(0, 9) select n * i++;
  var q2 = from n in Enumerable.Range(0, 9).AsParallel() select n * j++;
  foreach (var n in q1) Console.WriteLine(n + " ");
  Console.WriteLine();
  foreach (var n in q2) Console.WriteLine(n + " ");
  Console.WriteLine();
}
```



As shown above, the result of the last query is not correct (the result of the last query will vary from time to time).

The first query is a sequential query that performs the calculations using a local variable. For each iteration of the query, the local variable is updated, but it presents no problems as the expression is executed sequentially. The second expression is in principle identical, but it is executed as a parallel query. The result is that there are now several threads that unsynchronised update the local variable *j*, and one can thus not know (assume) what value *j* has. The problem can of course be solved in many ways, but the important thing is to be careful with side effects associated with PLINQ.

Paralleling a task with PLINQ is simple, as everything happens automatically, both splitting the task into partitions and subsequent combination the results. However, there are a number of possibilities for intervening in how the parallelization takes place, but it is outside the scope of this book.

7.4 THE CLASS PARALLEL

.NET has other options for parallel programming than PLINQ. The Parallel class has three static methods:

1. *Parallel.Invoke* which is used to perform several methods in parallel.
2. *Parallel.For* performing a usual *for* loop in parallel
3. *Parallel.ForEach* which in the same way performs a usual *foreach* loop in parallel

All three methods block until all work is completed.

The static method *Parallel.Invoke()* has the following signature:

```
void Parallel.Invoke(params Action[] actions)
```

and it executes an array of *Action*'s in parallel, where the parameter refers to the delegate *System.Action*. The method waits until all actions are completed.

This example *ParallelSorting* starts three sorting methods, each of which sorts an *int* array of 50,000 elements, but such that the three sorting methods are started in parallel and executed on their respective CPUs. The sorting methods are the three classic sorting methods, which are simple but not very effective:

```
public static class Sort
{
  public static void BubbleSort<T>(T[] t) where T : IComparable<T>
  {
    for (int j = t.Length - 1; j >= 0; --j)
    {
      bool ok = true;
      for (int i = 0; i < j; ++i)
        if (t[i].CompareTo(t[i + 1]) > 0)
        {
          Swap(ref t[i], ref t[i + 1]);
          ok = false;
        }
      if (ok) break;
    }
  }

  public static void SelectionSort<T>(T[] t) where T : IComparable<T>
  {
    for (int i = 0; i < t.Length - 1; ++i)
    {
      int k = i;
      for (int j = k + 1; j < t.Length; ++j) if (t[k].CompareTo(t[j])
      > 0) k = j;
      if (k != i) Swap(ref t[i], ref t[k]);
    }
  }

  public static void InsertionSort<T>(T[] t) where T : IComparable<T>
  {
    for (int i = 1; i < t.Length; ++i)
    {
      bool ok = false;
      for (int j = i; j > 0 && !ok; --j)
        if (t[j].CompareTo(t[j - 1]) < 0) Swap(ref t[j], ref t[j - 1]);
        else ok = true;
    }
  }

  public static void Swap<T>(ref T t1, ref T t2)
  {
    T t = t1;
    t1 = t2;
    t2 = t;
  }
}
```

I will not explain the three methods here, but the point is that these are methods that have taken time, and for large arrays, they even take a very long time.

Next, there is the program itself:

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

  static void Main(string[] args)
  {
    int[] v1 = Create();
    int[] v2 = Clone(v1);
    int[] v3 = Clone(v1);
    Parallel.Invoke(() => SortAction(Sort.BubbleSort, v1, "Bubblesort"),
                    () => SortAction(Sort.SelectionSort, v2,
                    "Selectionsort"),
                    () => SortAction(Sort.InsertionSort, v3,
                    "Insertionsort"));
    Console.WriteLine("Ok");
  }

  static void SortAction(Action<int[]> sort, int[] v, string name)
  {
    Stopwatch sw = new Stopwatch();
    sw.Start();
    sort(v);
    sw.Stop();
    Console.WriteLine(name + ": " + sw.Elapsed.TotalMilliseconds);
  }

  static int[] Create()
  {
    int[] arr = new int[50000];
    for (int i = 0; i < arr.Length; ++i) arr[i] = rand.Next(10000, 100000);
    return arr;
  }

  static int[] Clone(int[] arr)
  {
    int[] v = new int[arr.Length];
    Array.Copy(arr, v, arr.Length);
    return v;
  }
}
```

If you run the program, you will find that the three sorting methods are performed in parallel. The result could for example be:

where the three numbers indicate how long the sorting has taken measured in milliseconds. The last Ok is to show that *Parallel.Invoke()* is waiting (blocked until) all methods are complete.

The method *Create()* creates an array of 50,000 numbers and does not require any special explanation. The method *Clone()* creates a copy of an array, and it is used to ensure that it is the same array that the three methods sort.

Then there is the method *SortAction*, which is an *Action* with three parameters. The first parameter is an *Action* and refers to a sort method. The next parameter is the array to be sorted. Finally, the last parameter is just a text that the method prints. The method starts by setting up a stopwatch to measure how long the sorting takes. After the stopwatch has started, the sorting is performed, and finally how long the sorting has taken is printed.

Then there is the *Main()* method, which creates an array and clones it twice. Next, start three sorts with three *SortAction* objects. The result is that the three methods are performed in parallel.

Parallel.For are in principle identical to the corresponding loop construction in C#, but each iteration is performed in parallel rather than sequentially. The syntax is:

```
ParallelLoopResult For (int fromInclusive, int toExclusive, Action
<int> body)
```

The following example prints all numbers greater than or equal to 1000 and less than 1100 so that the program prints both the number and its cross sum:

```
class Program
{
  static void Main(string[] args)
  {
    Parallel.For(1000, 1100, n => Print((ulong)n));
  }

  static void Print(ulong t)
  {
    Console.WriteLine("{0, 10} {1}", t, CrossSum(t));
  }

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

The syntax is easy enough to understand, and the system will find out for itself how the task should be divided into partitions, which can be run in parallel. If you run the program, you will see that you can not predict which numbers will be processed first, as it happens in parallel on several CPU cores.

This method *Parallel.ForEach()* corresponds to *foreach*, and in the same way as *Parallel.For()* above, the difference is that the system performs the individual iterations in parallel. The syntax is:

```
ParallelLoopResult ForEach <TSource>
  (IEnumerable <TSource> source, Action <TSource> body)
```

The following program performs exactly the same as above, only this time using a *Parallel.ForEach()*:

```
class Program
{
  static void Main(string[] args)
  {
    Parallel.ForEach(Enumerable.Range(1000, 1100), t => Print((ulong)t));
  }

  static void Print(ulong t)
  {
    Console.WriteLine("{0, 10} {1}", t, CrossSum(t));
  }

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

PROGRAM 1: MATRIX MULTIPLICATION

A matrix is numbers arranged in 2-dimension array, for example

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

which is a matrix with 3 rows and 4 columns. If so one says that it is a 3x4 matrix. If you have two matrices where the number of columns in the first matrix is the same as the numbers of rows in the second matrix you can create the product of the two matrices. If you have a matrix and a matrix you can calculate there product which is a matrix where the element with index is the sum of all products of elements in row *r* in the first matrix and column *c* in the second matrix. If for example you have two matrices

$$A = \begin{Bmatrix} 2 & 5 & 6 & -2 \\ 8 & -3 & 9 & 1 \\ 5 & -4 & 4 & 8 \end{Bmatrix}$$

$$B = \begin{Bmatrix} 9 & 5 & -2 \\ 4 & 5 & -7 \\ 1 & -3 & -3 \\ 8 & 1 & 6 \end{Bmatrix}$$

there product is a matrix:

$$C = \begin{Bmatrix} 28 & 15 & -69 \\ 77 & -1 & -16 \\ 97 & 1 & 54 \end{Bmatrix}$$

In this exercise you should write a program that can calculate the product of two matrices. Create a new console application which you can call *Matrices*. Add a class *Matrix*:

```
class Matrix
{
  private double[,] value;

  public Matrix(int rows, int cols) { ... }

  public int Rows { get; private set; }
  public int Cols { get; private set; }

  public double this[int r, int c]
  {
    get { return value[r, c]; }
    set { this.value[r, c] = value; }
  }

  // Initialize this matrix with random numbers between a and b.
  public void Init(Random rand, double a, double b) { ... }

  // Print this matrix on the console.
  public void Print(int w, int d) { ... }

  // Create and return the product of this matrix and the matrix B.
  public Matrix Mul1(Matrix B) { ... }

  // Calculate the element with index (r,c) in the product matrix.
  private void Calculate(Matrix B, Matrix C, int r, int c) { ... }
}
```

Create a test method

```
static void Test0()
{
  // create the two matrices shown above
  // print the two matrices
  // print the product of the two matrices
}
```

When you have big matrices with many rows and many columns it can take a long time to calculate there product. You should then try to and a method *Mul2()* to the class *Matrix*, when the method must have the same prototype as *Mul1()* and when the method should create the product using *Parallel.For()*.

Create another test method which creates two matrices with values between -10 and 10, where the first has three rows and four columns while the other has four rows and five columns. Print the two matrices as well as there product both using *Mul1()* and *Mul2()*. You can use the method to test that the method *Mul2()* works correct.

Write a third test method:

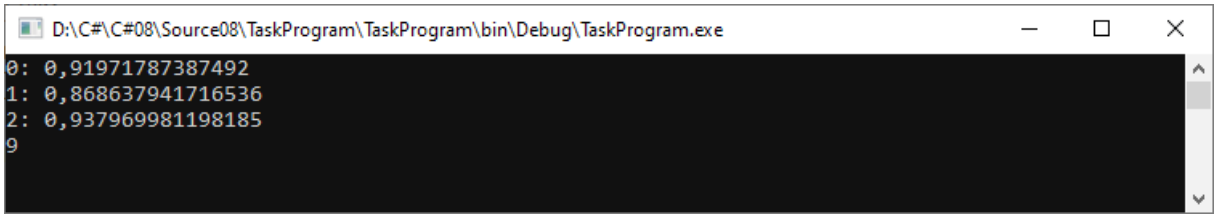
```
static void Test3(int m, int k, int n)
{
  // create a stopwatch
  // create a matrix with m rows and k columns
  // create a matrix with k rows and n columns
  // initialize the two matrices with random value
  // create the product of the two matrices using Mul1() and print
     the number
  // of milliseconds for the calculation
  // create the product of the two matrices using Mul2() and print
     the number
  // of milliseconds for the calculation
}
```

7.5 THE TASK CLASS

There is also a class called *Task* which can be used to start more threads in parallel, and the primary purpose is to make it easier to start more threads than by directly creating threads like Chapter 2. The class has many methods and the following is only a brief introduction. The program *TaskProgram* shows four simple examples. The first example creates and starts 3 tasks:

```
static void Test1()
{
  Task<Tuple<int, double>[]> tasks = new Task<Tuple<int, double>[]>(() =>
  {
    var results = new Tuple<int, double>[3];
    new Task(() => { Thread.Sleep(rand.Next(5000, 10000));
    results[0] = new Tuple<int, double>(0, rand.NextDouble()); },
    TaskCreationOptions.AttachedToParent).Start();
    new Task(() => { Thread.Sleep(rand.Next(5000, 10000));
    results[1] = new Tuple<int, double>(1, rand.NextDouble()); },
    TaskCreationOptions.AttachedToParent).Start();
    new Task(() => { Thread.Sleep(rand.Next(5000, 10000));
    results[2] = new Tuple<int, double>(2, rand.NextDouble()); },
    TaskCreationOptions.AttachedToParent).Start();
    return results;
  });
  Stopwatch sw = new Stopwatch();
  sw.Start();
  tasks.Start();
  tasks.ContinueWith(t => { foreach (Tuple<int, double> r in t.Result)
    Console.WriteLine(r.Item1 + ": " + r.Item2); }).Wait();
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds / 1000);
}
```

The method defines an array of the type *Task<Tuple<int, double>>* and then an array for *Task* objects. The type parameter means that a task must return a value of the type *Tuple<int, double>*. The array is initialized in the constructor with three *Task* objects, where each task sleep a random period between 5 and 10 seconds after which the task returns an index and a random *double* value (just do do something). When the array is created, the tasks are started and the method *ContinueWith()* define a method which is performed when the three tasks are performed. The test method also starts a stopwatch to measure the time for performing the three tasks, and if you run the method the result could be:



The three task must minimum use 15 seconds and maximum 30 seconds when executed serially, but they have only used 9 seconds which tells that they are performed in parallel.

There is also a class *TaskFactory*, which can make it easier to define and start tasks and the above method could instead be written as follows:

```
static void Test2()
{
  Task<Tuple<int, double>[]> tasks = new Task<Tuple<int, double>[]>(() =>
  {
    var res = new Tuple<int, double>[3];
    TaskFactory tf = new TaskFactory(TaskCreationOptions.AttachedToParent,
      TaskContinuationOptions.ExecuteSynchronously);
    tf.StartNew(() => { Thread.Sleep(rand.Next(5000, 10000));
      res[0] = new Tuple<int, double>(0, rand.NextDouble()); });
    tf.StartNew(() => { Thread.Sleep(rand.Next(5000, 10000));
      res[1] = new Tuple<int, double>(0, rand.NextDouble()); });
    tf.StartNew(() => { Thread.Sleep(rand.Next(5000, 10000));
      res[2] = new Tuple<int, double>(0, rand.NextDouble()); });
    return res;
  });
  Stopwatch sw = new Stopwatch();
  sw.Start();
  tasks.Start();
  tasks.ContinueWith(t => { foreach (Tuple<int, double> r in t.Result)
    Console.WriteLine(r.Item1 + ": " + r.Item2); }).Wait();
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds / 1000);
}
```

The class *TaskFactory* can also be used to transfer a parameter to a task. The type of the parameter is *object* and as so it is necessary with a type cast. The next method performs the same as the above methods, but this time the interval for how long the tasks should sleep is a parameter:

```
static void Test3()
{
  Task<Tuple<int, double>[]> tasks = new Task<Tuple<int, double>[]>(() =>
  {
    var res = new Tuple<int, double>[3];
    TaskFactory tf = new TaskFactory(TaskCreationOptions.AttachedToParent,
      TaskContinuationOptions.ExecuteSynchronously);
    tf.StartNew((obj) => { Tuple<int, int> t = (Tuple<int, int>)obj;
      Thread.Sleep(rand.Next(t.Item1, t.Item2));
      res[0] = new Tuple<int, double>(0, rand.NextDouble()); },
      new Tuple<int, int>(5000, 10000));
    tf.StartNew((obj) => { Tuple<int, int> t = (Tuple<int, int>)obj;
      Thread.Sleep(rand.Next(t.Item1, t.Item2));
      res[1] = new Tuple<int, double>(0, rand.NextDouble()); },
      new Tuple<int, int>(4000, 12000));
    tf.StartNew((obj) => { Tuple<int, int> t = (Tuple<int, int>)obj;
      Thread.Sleep(rand.Next(t.Item1, t.Item2));
      res[2] = new Tuple<int, double>(0, rand.NextDouble()); },
      new Tuple<int, int>(6000, 8000));
      return res;
  });
  Stopwatch sw = new Stopwatch();
  sw.Start();
  tasks.Start();
  tasks.ContinueWith(t => { foreach (Tuple<int, double> r in t.Result)
    Console.WriteLine(r.Item1 + ": " + r.Item2); }).Wait();
  sw.Stop();
  Console.WriteLine(sw.ElapsedMilliseconds / 1000);
}
```

The class *TaskFactory* has many properties and if you do not need these properties you can also write something like the following:

```
static void Test4()
{
  Task<Tuple<int, double>>[] tasks = new Task<Tuple<int, double>>[3];
  tasks[0] = Task.Run(() => { Thread.Sleep(rand.Next(5000, 10000));
    return new Tuple<int, double>(0, rand.NextDouble()); });
  tasks[1] = Task.Run(() => { Thread.Sleep(rand.Next(5000, 10000));
    return new Tuple<int, double>(0, rand.NextDouble()); });
  tasks[2] = Task.Run(() => { Thread.Sleep(rand.Next(5000, 10000));
    return new Tuple<int, double>(0, rand.NextDouble()); });
  Task.WhenAll(tasks).ContinueWith(t => { foreach (Tuple<int, double> r in
    t.Result) Console.WriteLine(r.Item1 + ": " + r.Item2); });
}
```

8 CALENDAR

As a final example, I will show the development of a program that can display a calendar on the screen, but in addition, the program should keep track of appointments and the like. There are many such programs for any platform, and the goal of the following is primarily to write a program that uses multiple threads, but perhaps also a program with some practical value. The program is not quite simple as it uses relatively complex algorithms.

8.1 TASK-FORMULATION

The task is to write a program that in a window can display a calendar. The window should show the calendar for a month and it should be possible to navigate the calendar, for example next month, next year and so on. It should also be possible to enter a specific year and a month and then go directly to that month. The program shall cover the period from year 0 to year 9999, and the program must take account of the shift from the Julian calendar to the Gregorian calendar. The program should also be able to save a calendar for a selected period to a text file.

The program should be used as a daily calendar program, and one should therefore be able to enter notes and appointments, set alarms and set special (custom) anniversaries beyond the days that the calendar was born with (holidays). The calendar must give a warning, when the time for an appointment occurs.

8.2 ANALYSIS

The analysis will consist of

1. Requirements specification
2. prototype for the user interface
3. development plan

Requirements specification

The main application window should primarily show a calendar for a specific month.

The calendar will cover the period from year 0 to year 9999 and should take account of the shift from the Julian to the Gregorian calendar. The program should take into account the fact that the switch to the Gregorian calendar has not happened the same year in all countries, and as such the time for the shift to the Gregorian calendar must be an option.

For each date, the calendar must display the following information:

- the day's number in the month
- the day's name in the week
- the week's number in the year, if it is a Monday
- an indication if it is a public holiday or anniversary
- a mark when notes are created for that day
- a mark when appointments are created for that day
- a mark when anniversaries are created for that day

In addition, the program must show:

- When the sun rises and when the sun goes down
- Phases of the Moon

For the holidays, the program must be programmed to (know) the following days:

- New Year's Day
- Palm Sunday
- Maundy Thursday
- Good Friday
- Easter Sunday
- Easter Monday
- Prayer Day
- Christ's Ascension
- Pentecost
- Whit Monday
- Christmas Eve
- Christmas Day
- Second Christmas Day

and also the program must be able to show custom anniversaries (for example the wife's birthday). An anniversary falls on a specific date and every year after the introduction of the Gregorian calendar (with the option to specify a start and end year). If an anniversary specifies d. 29/2, and it is not a leap year, the date 1/3 should be used.

The program distinguishes between notes and appointments. A note is any text entered for a specific date, and there can be more notes for the same date. An appointment is a short message relating to a specific time (clock) and the program must be able to notify the user automatically, when the time for an appointment occurs. Notes will be saved until the user manually delete them. Appointments should be automatically deleted, when the time is exceeded, but maybe there should be an option, that an appointment should be stored after the time is exceeded as a form of history.

Regarding appointments it must be able to make an appointment for a specific date and for a specific time. There may then be several appointments the same day, and an appointment is in principle only plain text. An appointment can span multiple days. An appointment can also have a start time, an end time, or both, and if so, the program should come with a warning if times conflicts. When the time of an appointment occurs, the program must come with a warning that must appear in the user interface. It is adopted

1. the warning should appear at the day's start (when the computer is turned on)
2. 1 hour before the appointment occurs
3. 15 minutes before the appointment occurs
4. 5 minutes before the appointment occurs
5. when the appointment occurs, and the appointment should be deleted

Functions

Navigate the calendar:

- Shift to previous year
- Shift to previous month
- Shift to next month
- Shift to next year
- Enter year and month and the calendar must shift to that month

The program must be able to save a calendar as a comma separated text file with the following format:

```
year; month number; month name; day in month; day name; week number
[; holiday]
```

and an example could be

```
2012;4;April;1;Sunday;14;Palm Sunday
2012;4;April;2;Monday;14;
2012;4;April;3;Tuesday;14;
```

The program must also have the following features:

- Maintenance of anniversaries
- Maintenance of notes
- Maintenance of appointments
- Watch
- Alarm (that comes with a warning a certain time)
- Timer (that comes with a warning when a timer reach 0)
- Stopwatch

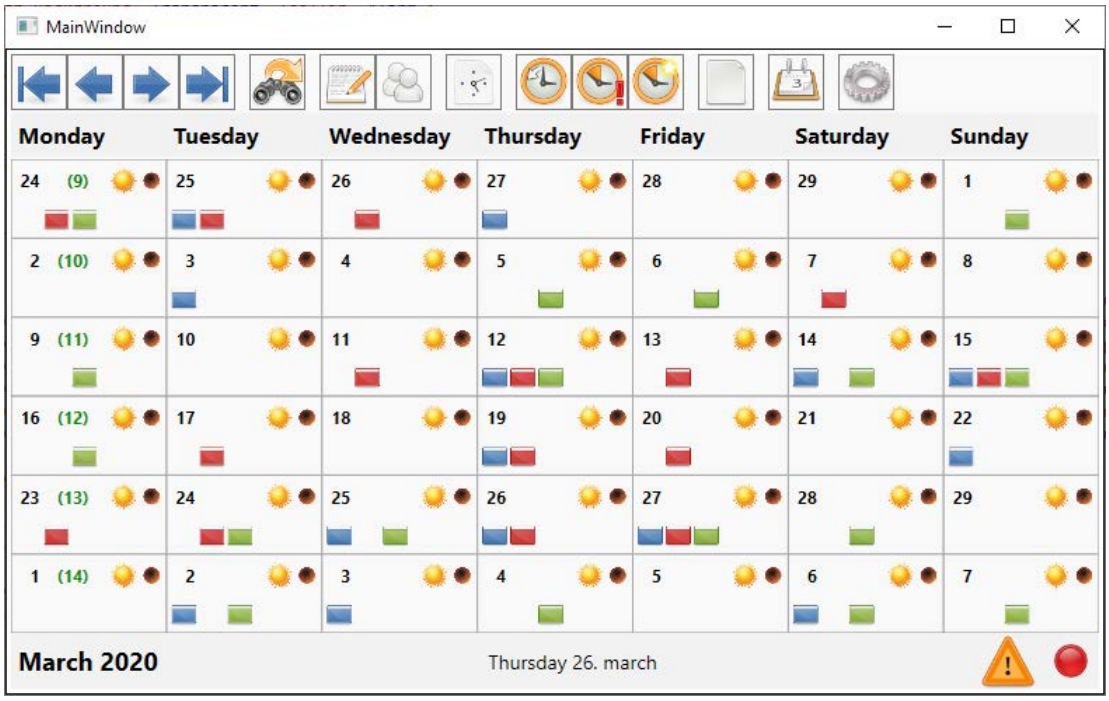
The prototype

The prototype is a Visual Studio project called *Calendar*. The result is a program that only opens the window below. All the program's functions are accessible from a toolbar at the top of the window:

1. The first four icons (arrows) are used to navigate the calendar.
2. The next opens a dialog box so you can navigate to a specific month.
3. The sixth icon opens a dialog to maintenance notes.
4. The seventh icon opens a dialog to maintenance appointments.
5. The next shows the clock (how the clock should look like is not yet determined).
6. The next again is for the stopwatch. Opens a dialog box to start or stop the stopwatch.
7. The tenth icon opens a dialog box to maintenance of alarms.
8. The eleventh icon opens a dialog box to maintaining timers.
9. The next icon is to export of a calendar to a text file and opens a dialog box for selecting the period.
10. The second last icon opens a dialog box to maintenance of anniversaries.
11. The last icon is the settings for the program, but it has not yet determined, which settings should available, but possibly colors.

At the bottom is a status bar showing

- the current month
- the current day (today)
- an icon that shows whether there is set an alarm (if not, no icon appears)
- an icon that shows whether there is set a timer (if not, no icon appears)



For each date is shown the day's number in the month (and for Monday the week number). In addition, there are five icons, which means that there are

1. notes for that date
2. appointments for that date
3. anniversaries of that date
4. The moon's phase
5. When the sun rises and when the sun goes down

In the prototype each day is a user control that must be further developed later. If you click on an icon, you the program must show a simple dialog box with information, and for appointments it should also be possible to cancel the appointment. If you right click on a specific date, you should get a popup menu where you can

1. create a note
2. create an appointment
3. create an anniversary

The development plan

The project is developed through the following iterations:

1. The algorithms
2. Design
3. Navigate the calendar
4. Maintenance of notes
5. Appointments and anniversaries
6. Implementation of all functions relating to the watch and including alarm and timer
7. Implementation of export of a calendar
8. A last iteration and code review

8.3 THE ALGORITHMS

Implementing the calendar algorithms is is the most comprehensive of the above iterations and it is also the most complex. Algorithms for manipulating the calendar are generally relatively complex, and in this case, there are three issues:

1. to work with dates within the period years 0 to 9999
2. to determine the phases of the moon
3. to determine the height of the sun in the sky

As far as the first is concerned, the problem was that until 1582 the Julian calendar was used, after which it was changed to the Gregorian calendar, and the consequence is that in the Gregorian calendar you have to omit some days that the Julian calendar has calculated too much. The earliest use of the Gregorian calendar is October 1582, and this means that October 4 is followed by October 15. Thus, 10 days are missing. This is all complicated by the fact that not all countries has switched to the Gregorian calendar at the same time anywhere in the world. For example, in the United States, the Gregorian calendar was first used by all states from August 6, 1900.

For these reasons it is relatively complicated to write a calendar (and that is a date class) that runs correctly from year 0 to 9999, and thus also applies to historical periods, and which also takes into account when the Gregorian calendar is taken in use.

One of the requirements was that the calendar should support a calendar that goes from year 0 to 9999. One can discuss whether it makes sense. First, a calendar that goes back to year 0 is problematic, as the holidays are different if you go far back in time. Similarly,

there are problems with a calendar that extends all the way up to 9999. Firstly, the calendar will probably be changed before that time (even the Gregorian calendar is not perfect), and secondly, the algorithms regarding the positions of the sun and moon are not precise enough. Thus, one must expect uncertainty for dates that are many years ahead of our time. For these reasons, it is decided that the calendar should only span the period from 600 to 3199, and for all uses of the program, it will also be fully adequate. For dates before that time, is for example, the times of Easter not fixed and depend on where one is in the world. Similarly, one must expect the calendar after the year 3200 has to be modified, as it has moved one day.

As mentioned, C# has a *DateTime* class for date and time, so it is natural to be interested in whether it can be used. The class have essentially the properties needed, but they do not support the Julian calendar, and if it is maintained that it is part of the task, this class cannot be used directly. I have therefore decided to write my own date class, which should have the same properties as *DateTime*. Below I will give a brief presentation of two classes *Calendar* and *CalendarTime*, which works in the same way as *DateTime*, but also supports the Julian calendar.

For the last two problems, the challenge is the same, namely to implement several algorithms regarding the rotation the moon and the earth's rotation around the sun. These algorithms are based on the geometry of the sphere and a number of physical laws. I will not deal with these algorithms here, but only give a brief summary, but there are many examples on the Internet that show how the algorithms can be implemented. When I do not want to deal with them here it is because they are complex and a complete review will fill much and fall outside the subject of these books, and here reference is made to classic literature on astronomy.

To implement the algorithms I have created a class library project corresponding to the fact that the implementation of the algorithms has their own interest to some extent. The library contains five classes

1. *Calendar* which represents a calendar with cutover dates for the Julian calendar. The class also has important calendar methods and here methods to calculate to and from a Julian day.
2. *CalendarTime* which is the time class that in the same way as *DateTime* represents a time for date and count in milliseconds.
3. *Sun* which for a given time calculates the suns position and sun rise and set times.
4. *Moon* which looks like the class *Sun* and performs the same calculations, just instead for the moon.
5. *Tools* which is a source file containing some classes with mathematical material used by the classes *Sun* and *Moon*.

The project is called *CalendarTools*. The class *Calendar* is defined as follows:

```
public class Calendar : IComparable<Calendar>
{
  /// <summary>
  /// Represents the last date in the Julian Calendar on the format:
  YYYYMMDD.
  /// </summary>
  public int Date1 { get; private set; }

  /// <summary>
  /// Represents the first date in the Gregorian Calendar on the
  format: YYYYMMDD.
  /// </summary>
  public int Date2 { get; private set; }

  /// <summary>
  /// The number of days to skip in this calendar.
  /// </summary>
  public int Skip { get; private set; }
```

The class is relatively complex, but it represents a concrete calendar in the form of two dates (in the format YYYYMMDD), which indicates respectively the end of the Julian calendar and the start of the Gregorian calendar. In addition, the class has a variable that keeps track of how many days are skipped. I should not explain the class and the algorithms here, but when studying the code you should note:

- How there are three constructors, the first one being a default constructor, which creates a calendar with the transition to the Gregorian calendar on Oct. 4, 1582, while the next, as a parameter, has a *CultureInfo* object that determines the transition to the new calendar and number of days to be skipped. The class knows the start of the Gregorian calendar for a few countries, but for all others the default value is used. Finally, the last constructor allows for a custom calendar.
- The class has methods for determining general characteristics of the calendar and here especially if a date is legal. The class also has methods to determine the next day and the previous day.
- The class has a method that converts a date to its Julian day, which is a consecutive day number from noon the year 4712 B.C. There is a reverse method, that converts the other way and thus converts a Julian day to a corresponding date in the current calendar.
- A method *EasterDay()* that calculates the date of Easter Sunday for a given year.
- A method that calculates the week number for a date.

Then there is the class *CalendarTime*, which is a class that defines a time for a specific *Calendar* object:

```
public class CalendarTime : IComparable<CalendarTime>
{
  private long value;              // represents a date as YYYYMMDDHHMMSSMMM
  private Calendar calendar;
```

A date is represented as a *long* with fields for year, month, day, hour, minute, second and millisecond. The class is immutable and has no public constructor, but it has many static factory methods, which creates *CalendarTime* objects for a variety of parameters. The class fills a lot, but is in principle quite simple, since the class *Calendar* implements the necessary algorithms. You should especially note that the class has add methods

- *AddYears()*
- *AddMonths()*
- *AddDays()*
- *AddHours()*
- *AddMinutes()*
- *AddSeconds()*
- *AddMilliSeconds()*

which can have both positive and negative arguments and are used to return a time that is current time moved back or forth.

The algorithms for calculating the sun and the moon's position in the sky are implemented in two classes:

Sun
Moon

Both classes has a constructor with three arguments:

1. A *CalendarTime* object
2. Latitude
3. Longitude

and from these arguments the constructor initializes some properties for the sun or the moon. The algorithms are taken from a project that makes these algorithms available:

```
https://github.com/shred/commons-suncalc
```

and the algorithms are modified for this project, but all the important and many calculations are attributed to the above project.

The class *Sun* represents the following properties of the position of the sun:

- *Azimuth*, which is the direction along the horizon in degrees, measured from north to east (0 = north, 135 = southeast, 270 = west).
- *Altitude*, which is the visible altitude of the sun above the horizon in degrees (0 = the sun's center the horizon, 90 = zenith).
- *Distance*, which is the distance from the earth to the sun in kilometers.
- *SunRise*, which is the time for the next sunrise
- *SunSet*, which is the time for the next sunset
- *SunNoon*, which is the when the sun is highest in the sky

The class *Moon* represents the following properties of the position of the moon:

- *Azimuth*, which is he north-based azimuth of the moon in degrees (0 = north, 135 = southeast, 270 = west).
- *Altitude*, which is the altitude above the horizon, in degrees (0 = the moon's center is at the horizon, 90 = zenith).
- *Distance*, which is the distance from the earth to the moon in kilometers.
- *MoonRise*, which is the time for the next moon rise.
- *MoonSet*, which is the time for the next moon set.
- *Fraction*, which is the illumination of the moon (0 = new moon, 1 = full moon).
- *Phase*, which is the moon phase. Starts at -180 (new moon, waxing), passes 0 (full moon) and moves toward 180 (waning, new moon).
- *NewMoon*, which is the time for the next new moon.
- *FirstQuarter*, which is the time for the next first quarter.
- *FullMoon*, which is the time for the next full moon.
- *LastQuarter*, which is the time for the next last quarter.

Before uses these classes in the program they must be tested. This could be done by writing unit tests, but in this case I have chosen to write a test program instead. I don't want to show the program here, but the project is called *TestTools*.

8.4 DESIGN

The design phase is performed in four steps:

1. Design of the project architecture
2. Design of the model layer
3. Design of the DataAccess layer
4. Design of the user interface

The design starts to create a copy of the project from the analysis. The copy is called *Calendar1*.

Design of the architecture

The program's architecture has a classic MVVM architecture, and the project is expanded with folders for the usual layers.

The program must save information about notes, appointments and anniversaries and it is decided to store this information in a database with three tables. As it is a typical single user program, I will use simple file based database and I will use SQLite. Therefore, the necessary packages of assemblies has been added to the project using NuGet.

Design of the model layer

The program must maintain three entities and should therefor have three model classes:

1. *Note*, which is a text stored for a specific day
2. *Appointment*, which is a small message assigned for a specific time a day and may be a second time indicating the end of the appointment
3. *Anniversary*, which is a name for a day and can be a holiday

A note is quite simple and is defined by four properties:

```
public class Note
{
  public int Id { get; set; }
  public CalendarTime Date { get; set; }
  public string Title { get; set; }
  public string Text { get; set; }
}
```

An appointment is defined as:

```
public class Appointment
{
  public int Id { get; set; }
  public CalendarTime From { get; set; }
  public CalendarTime To { get; set; }
  public string Text { get; set; }
  public bool IsSaved { get; set; }
}
```

An appointment must have a *From* time, but *To* is optional. The last property is used to indicate where this appointment should automatic be removed if the time is passed.

An anniversary is defined as:

```
public class Anniversary
{
  public int Id { get; set; }
  public CalendarTime Date { get; set; }
  public int? AtEaster { get; set; }
  public int? From { get; set; }
  public int? To { get; set; }
  public string Text { get; set; }

  public bool IsHoliday { get; set; }
}
```

The last two properties are necessary and must have a value while exactly one of the two properties *Date* and *AtEaster* must have a value. If *Date* has a value it means the date for this anniversary, and if *AtEaster* has value it means the number of days (positive or negative) from Easter day for this anniversary. The two properties *From* and *To* can be used for two year numbers to indicate a period for this anniversary.

Design of the DataAccess layer

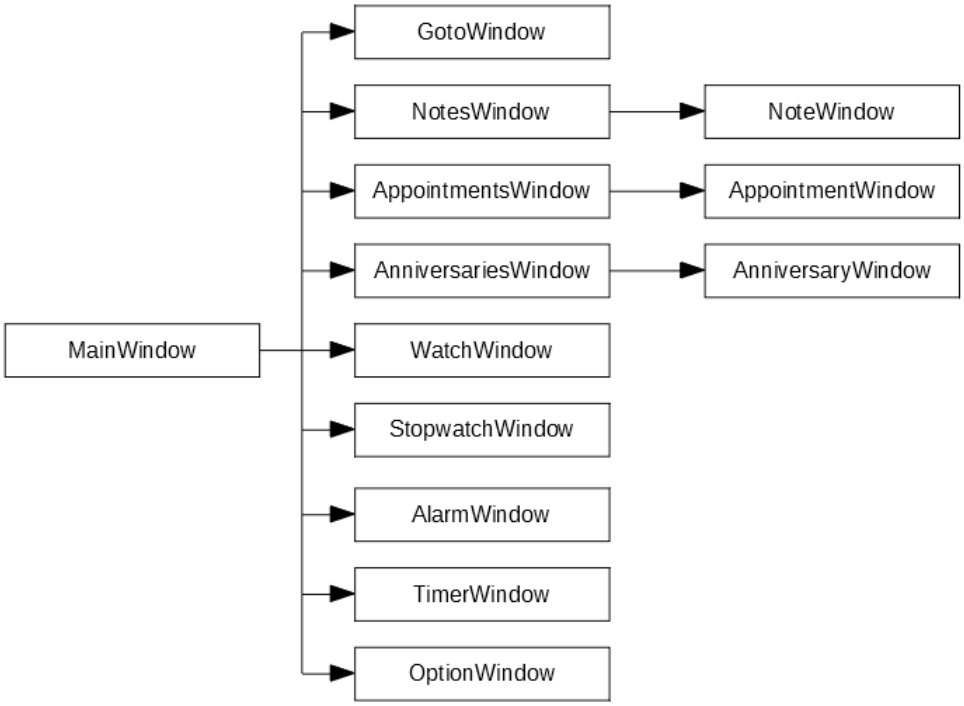
This layer contains two types, an interface *IDb* and a class implementing this interface. The interface is:

```
public interface IDb
{
  event EventHandler DatabaseCreated;
  event EventHandler<DBEventArgs<Note>> NotesChanged;
  event EventHandler<DBEventArgs<Appointment>> AppointmentsChanged;
  event EventHandler<DBEventArgs<Anniversary>> AnniversariesChanged;

  Note GetNote(int id);
  List<Note> GetNotes();
  List<Note> GetNotes(CalendarTime date);
  List<Note> GetNotes(CalendarTime date1, CalendarTime date2);
  void AddNote(Note note);
  void UpdateNote(Note note);
  void RemoveNote(Note note);
  void RemoveNotes(CalendarTime date);
  void RemoveNotes(CalendarTime date1, CalendarTime date2);
  Anniversary GetAnniversary(int id);
  List<Anniversary> GetAnniversaries();
  List<Anniversary> GetAnniversaries(string text);
  void AddAnniversary(Anniversary anniversary);
  void UpdateAnniversary(Anniversary anniversary);
  void RemoveAnniversary(Anniversary anniversary);
  void RemoveAnniversary(string text);
  Appointment GetAppointment(int id);
  List<Appointment> GetAppointments();
  List<Appointment> GetAppointments(CalendarTime date);
  List<Appointment> GetAppointments(CalendarTime date1, CalendarTime
  date2);
  void AddAppointment(Appointment appointment);
  void UpdateAppointment(Appointment appointment);
  void RemoveAppointment(Appointment appointment);
  void RemoveAppointments(CalendarTime date);
  void RemoveAppointments(CalendarTime date1, CalendarTime date2);
  void CreateDB(string filename);
}
```

The method names should explain what the method performs, and generally there is a method for each required database operation. The interface is implemented by the class DB, and it is a comprehensive class, but in principle simple as each method is nothing but an encapsulation of a SQL expression.

Design of the user interface

The main window is defined by the prototype. In addition, the user interface includes a series of dialog boxes corresponding to the functions of the program, as defined in the analysis. The following diagram shows an overview.



If, for example looking at *NotesWindow*, it is a dialog box that shows a list of all notes within a selected period. *NoteWindow* is an associated dialog box that is used to maintain a single note.

8.5 NAVIGATE THE CALENDAR

The iteration starts with creating a copy of the project from the design. The copy is called *Calendar2*. The task in this iteration is to implement the first five buttons in the toolbar, but first the *MainWindow* and here the user controls for the dates must be bound to properties in a model. The following is a short descripting of what performed in this iteration.

The project is expanded with a *ViewModels* folder.

A *ViewModel* class *NoteModel* is added. The class encapsulates the model class *Note* and prepares the class for the data binding and implements validation of properties. In exactly the same way is added wrapper classes *AppointmentModel* and *AnniversaryModel*. These classes are in fact not used in this iteration but should be used in later iterations.

A class *DateViewModel* is added as a *ViewModel* for a user control representing a date. The class represents a date to be shown in the calendar and must define properties to bind in the user control. It is a relative complex class, as the class for a given date must check where there is notes, appointments and anniversaries for this date. For that the class has three *OberservableCollection* objects that contains objects of the type *NoteModel*, *AppointmentModel* and *AnniversaryModel*. These collections is reality not used in this iteration, but should be used later.

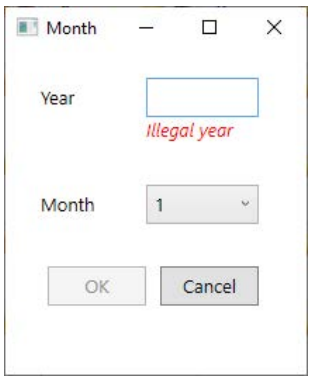
Code behind for the user control *DayControl* is changed. All code (all properties) are removed and two converters are added. The XML is changed such the controls are bound to properties in ViewModel class *DateViewModel* and the background color is also bound to a property in the same class.

A class *MainViewModel* is added as a ViewModel for the *MainWindow*. I has a 2-dimensional array of *DateViewModel* objects which are created when an instance of the class is created. The class creates commands objects for the five buttons in the tool bar. The class an instance variable for the month to show, and to navigate the calendar this variable must be updated and the the *DateViewModel* object in the array must be updated.

Code behind for *MianWindow* is changed. The window's *DataContext* is bound to an instance of *MainViewModel* and the method which creates the user controls and add these controls to the user interface is changed. When a user control is created it is bound to a *DateViewModel* object from the array in *MainViewModel*.

To *ViewModels* is added a class *Converters*, which defines converters for *MainWindow*, but the class should also contains converters used in other dialog boxes.

The XML for *MainWindow* is updated. The components in the status bar are bound to properties in *MainViewModel* and the first five buttons in the toolbar are bound to *Command* objects in *MainViewModel*. When you then run the program it shows a calendar for the current month and you can navigate the calendar, at least using the first four buttons. The last one requires a little more as it must open a dialog box where the user can select a year and a month. If the user clicks on this button the program opens the following window:



To implement this function a class *GotoView* is added to the *Views* folder, a class that defines the layout for the above window.

To the model layer is added a class *CurrentCalendar* representing a *Calendar* object from the class library. This class must be updated later when the options for the program is implemented, but for moment the class represent a default calendar, and the class is implemented as a singleton. The calendar object is needed to select the first day in a month, which in principle need not be 1.

To the ViewModels folder is added a class *GotoViewModel* that represents a model for *GotoView*. The class is defined as other ViewModel classes.

Code behind to *GotoView* is changed and bound to a *GotoViewModel* object the window's *DataContext*. The controls in XML is bound to properties, and the dialog box is ready for use.

Then the implementation of this iteration is finished and can be tested. It means you can run the program and navigate the calendar. Here you have to test especially what happens if you navigate to and from the shift from the Julian to the Gregorian calendar, which for the default calendar is October 1582.

8.6 MAINTENANCE OF NOTES

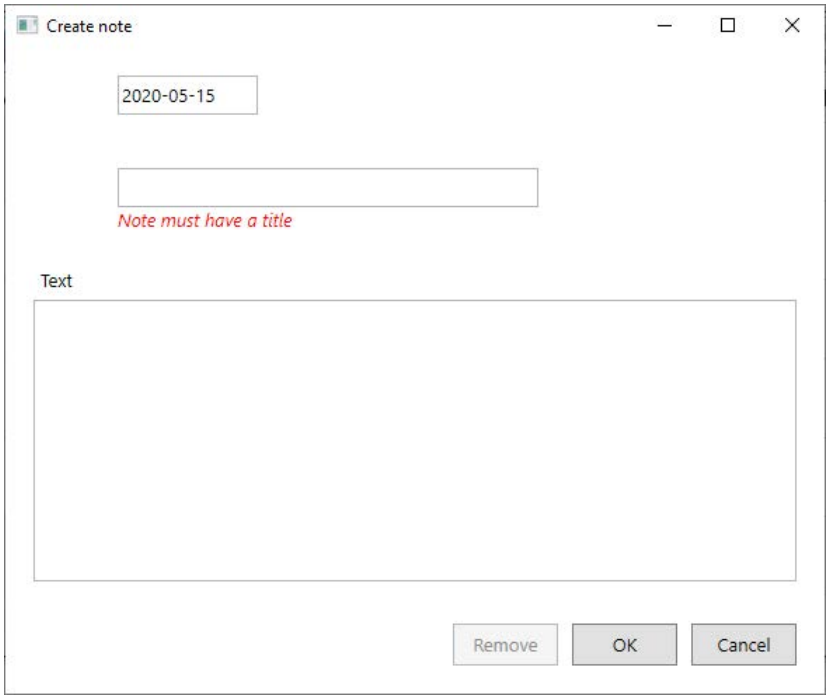
This iteration includes:

1. Create a new note by right click on a date in the calendar
2. Show all notes and maintenance of notes when click the note icon in the toolbar

The iteration starts with creating a copy of the project. The copy is called *Calendar3*. The iteration includes the following steps:

The class *NoteModel* is renamed to *NoteViewModel*.

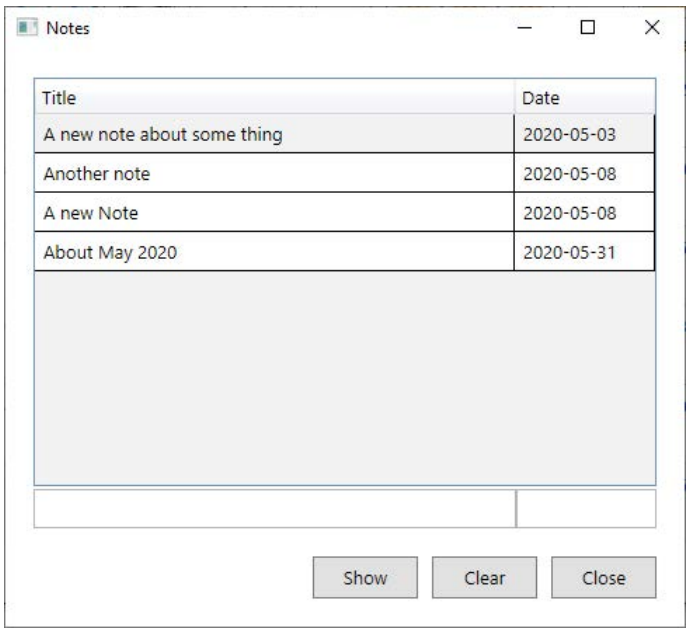
A new window *NoteWindow* is added to the *Views* layer:



The window should be used to create a new note, but also to edit an existing note. The model for this window is *NoteViewModel*, and the class is expanded with two new properties to bind the text in the title line as well as the visibility of the *Remove* button. The class is also expanded with commands for the three buttons.

The user control is expanded with a context menu with three menu items for create a note, an appointment and an anniversary. In this iteration only the first menu item is used. A user controls small icons for for respectively notes, appointments and anniversaries. The icon for notes is wrapped in a *Button* element as it must be possible to bind to a command.

If the user click in the toolbar on the *Notes* icon the program opens the following window:



which in a *DataGrid* with a filter shows all notes. The *Clear* button clears the filter, and the *Show* button opens the window shown above initialize with the selected note. The window has a *NotesViewModel* model where the notes are represented in an *ObservableCollection* transmitted as a parameter to the constructor.

The ViewModel class *DateViewModel* is updated with two new commands. If you right click on a day and opens the context menu and select *Create Note* the dialog box to create a new note opens. The other command is used if the user click for a date clicks on the icon for notes. If there is only one note the dialog box to edit the note opens, but else the above dialog box opens, but shows only the notes for the actual date.

There are some other important adjustments. The model for the above dialog box showing the notes is observer for an event fired by the dialog box for modify a note. The event is fired when a note is removed. In the same way is *MainViewModel* observer for events fired by the data access layer when the state of the database is changed.

8.7 APPOINTMENTS AND ANNIVERSARIES

As before the iteration starts with creating a copy of the Visual Studio project from the previous iteration. The copy is called *Calendar4*. The iteration consists of implementing of two functions, both of which are similar and both of them similar to the previous iteration.

Anniversaries

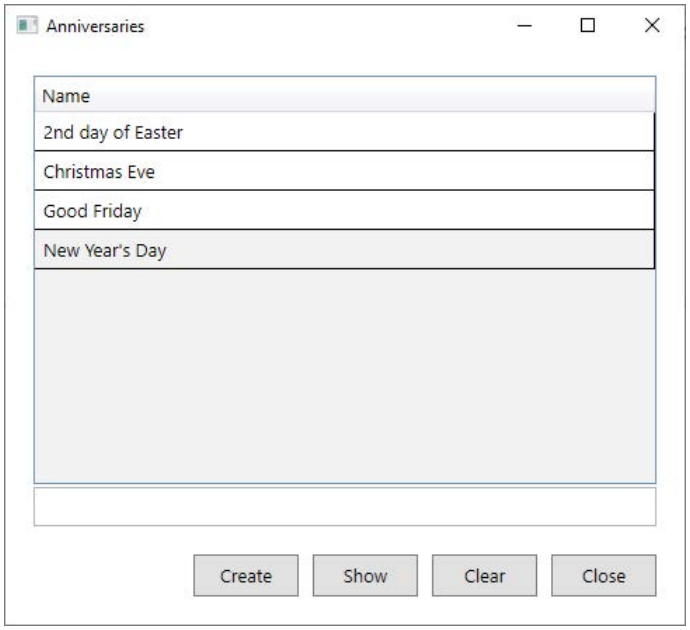
To implement maintenance of anniversaries a change of the model is required. An anniversary has a field for a date to denote the date for this anniversary. This field does not really make sense since an anniversary does not fall on a specific date in a particular year, but instead on a specific day in a month. In the database and similarly in the model class *Anniversary*, it has therefore been decided to change the *Date* field to a field *Month* and the corresponding the field *AsEaster* to a field *Day*. Month may be null, while *Day* must have a value. The interpretation is:

1. If *Month* has a value (between 1 and 12) *Day* must have a legal day number for this month. It has been decided that February 29 is not allowed.
2. If *Month* is null the value of *Day* must have a value between -365 and 365 and should denote the number of days from Easter for this anniversary.

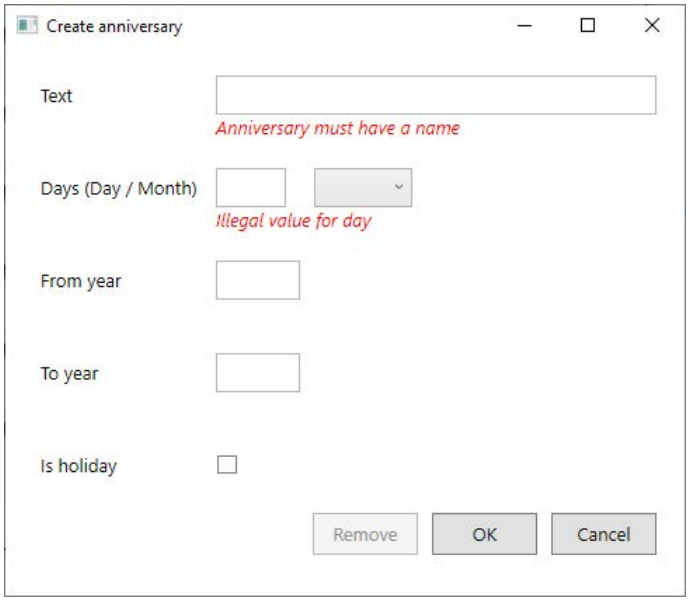
This change means that both the class *DB* and the class *Anniversary* must be updated, but also the class *AnniversaryModel* must be changed and the database must be created again. When the class is changed it is renamed to *AnniversaryViewModel*.

As another change, the menu item for anniversaries is removed from the context menu, since it doesn't really make sense to create an anniversary for a specific day. Instead, the feature has been moved to the *AnniversariesWindow* window.

Otherwise, the function is implemented in exactly the same way as for notes, and I will not explain how the function is implemented. If you click the icon in the toolbar the program opens the following window:



which shows a *DataGrid* with a filter, and if you click *Create* or click *Show* to the show the selected anniversary the program opens the window *AnniversaryWindow* and below used to create a new anniversary:



Appointments

To implement appointments the program must be extended with two windows which look likes the above, but instead used to show an overview over all appointments and to edit an appointment.

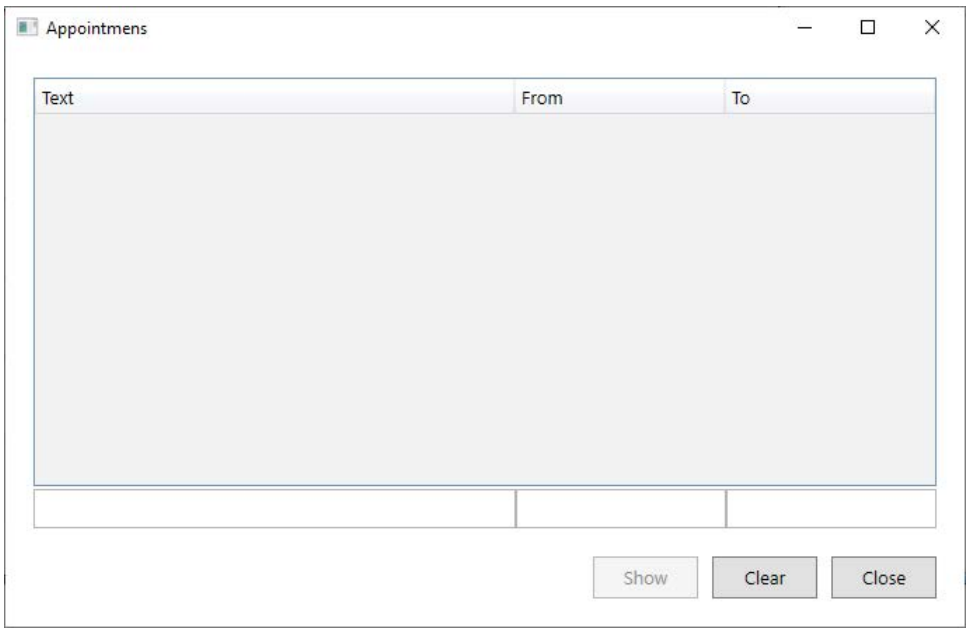
As the first the class *AppointmentModel* is renamed to *AppointmentViewModel*.

As with anniversaries, a change has been made. An appointment has a field to indicate where to save an appointment after the time has come. It has been decided to remove this field and leave it a setting that can be turned on and off. Currently, the function is implemented as if the setting is off and appointments are automatically deleted when the time expires. This change causes the following changes:

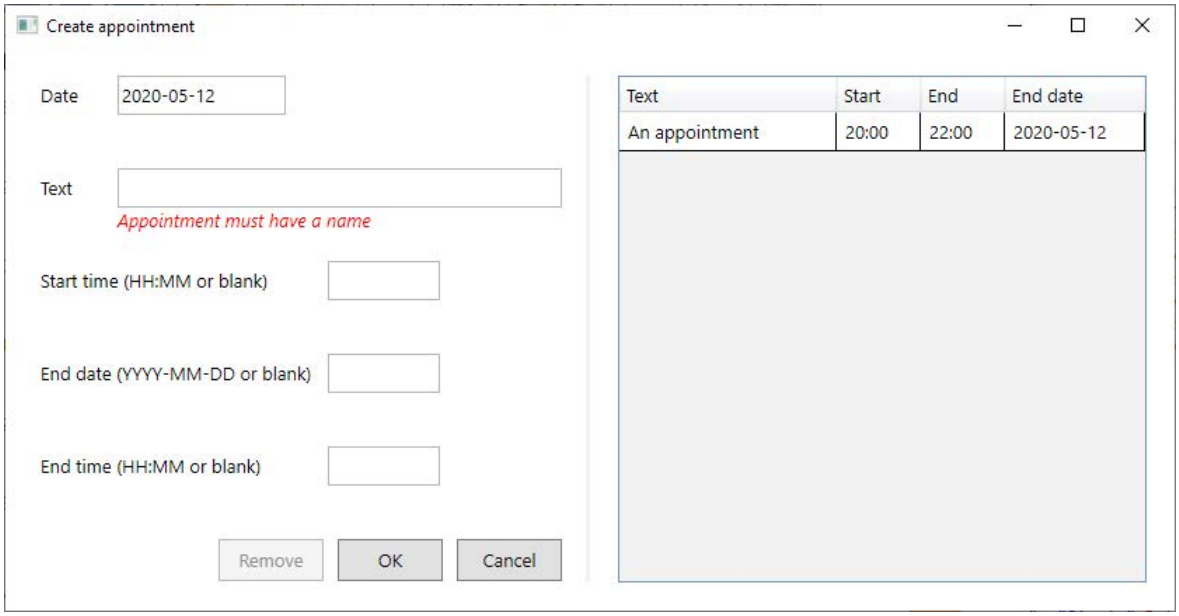
1. A property is removed from the model class *Appointment*.
2. Most methods in the class *DB* concerning appointments must be changed as the property no longer exists.
3. The class *AppointmentViewModel* is changed for the same reasons.

The column in the database indicating where to save an appointment is not removed as it should be used later when the settings is implemented.

When you click the button *Appointments* in the toolbar or the icon showing appointments for a date, the program opens the window below. The window shows all appointments with a filter. For each appointment the window shows the text and the period for the appointment. When you create an appointment from for context menu for a date *From* is automatic initialized to that date, but you can (and will typically do so) enter a start time. You can also enter and end date and an end time. If you do not enter an end date, the end date is initialized to the same date as the start date.



The window to create / modify an appointment is:



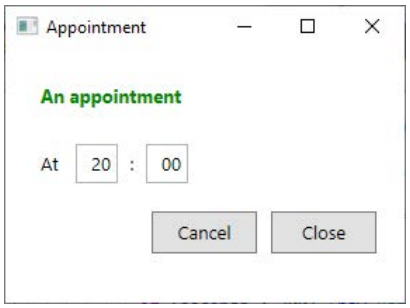
The *DataGrid* is used to show an overview over other appointments the same day. When you save the appointment the program checks if there is a conflict and if so give the user a warning. You should note that if you do not enter a start time or an end time the appointment will probably conflicts with all other appointments that day. You should also note, that you cannot create appointments for a date older than the current date.

Up to this point, the appointments function has in principle been implemented in the same way as the notes and anniversaries functions. However, appointments have two other features.

When you opens the program, it performs a method (implemented in *MainViewModel*) which

1. Remove all appointments with a *From* date older than the current date. The goal is to ensure that old appointments are automatically removed, but the feature needs to be changed later after settings are implemented. To implements this feature is added a method to the class *DB*.
2. If there for the current day are appointments the program opens the above dialog box showing the appointments for these day.

The second feature must notify the user when it is time for an appointment. For now it is decided that user should get a notification:



The notification must occur if there is less than one hour for that appointment, again when there is less than 15 minutes (the text changes to blue) and finally if there is less than 5 minutes when the text appears as red. This numbers should later be changed to settings. To implement this feature *MainWindow* starts a thread, which each minute checks if there is a notification (for the current date) and if so opens a dialog box with the notification. The dialog box is opened as a modeless dialog box (and the program then opens more dialog boxes). The user can close the dialog box, or the user can click Cancel to delete the appointment. To ensure that this dialog boxes are closed they start a thread which close the dialog box after 10 seconds.

After the completion of this iteration, the three primary functions Notes, Anniversaries and Appointments are in principle complete. I will therefore end this iteration with a refactoring of these three functions and a test. As part of this ending, I will note which settings to implement later.

Refactoring

The refactoring starts to create a copy of the project. The copy is called *Calendar5* and is then the result of the project after this iteration. I start to create a new database so all data stored as part of the development is gone. In the refactoring I want to focus mainly on:

1. methods not used
2. variables not used
3. unfortunate names
4. code that is comment out
5. standardization of design
6. general code optimization
7. delete unused using statements
8. unused converters or converters which do the same
9. Check hard coded strings
10. other adjustments

Notes

There are following classes:

- NoteViewModel (ViewModels)
- NoteWindow (Views)
- NotesViewModel (ViewModels)
- NotesWindow (Views)

The result is only few changes, and the most important is closing *NotesWindow* when the list is empty.

Anniversaries

There are following classes:

- AnniversaryViewModel (ViewModels)
- AnniversaryWindow (Views)
- AnniversariesViewModel (ViewModels)
- AnniversariesWindow (Views)

The result is as for notes only few changes.

Appointments

There are following classes:

- AppointmentViewModel (ViewModels)
- AppointmentWindow (Views)
- AppointmentsViewModel (ViewModels)
- AppointmentsWindow (Views)
- ReminderViewModel (ViewModels)
- RemainderWindow (Views)

As a result of the refactoring I have noted the following setting that should be implemented in the last iteration:

1. The database name and then, where the database is stored and then the opportunity to select another database.
2. A function to create a database.
3. How long a the notification modeless dialog box should be open must be a setting
4. The three times (5, 15 and 60 minutes) for notifications for appointments should be settings
5. How often to check for appointments must be a setting
6. After testing appointments, it has been decided, that notifications for appointments should be an option that can be turned on and off.

Test

To test the three function I defines for each function a sequence of test cases, and if these tests are performed without errors, I will perceive the function as tested. If there is errors they must be removed and the test performed again.

Test Notes:

1. Open the program
2. Create a note for a day in the current day
3. Create a note for another day in the current day
4. Create two other notes for the same days but for a month a year later
5. Navigate the calendar a year back
6. Create two notes for the same day as the first note
7. Create a note for the same day as the second note
8. Navigate the calendar a year forward and check you see icons for two day with notes
9. Open one of the two notes (click the note icon) and modify the content
10. Open the note again and test the modifications still are there, remove the note
11. Select Notes from the toolbar and check there is 6 notes on the list
12. Check the filter
13. Opens the first note you have created and modify the note
14. Close the note list
15. Navigate the calendar a year back and the calendar should show the current month
16. Click on the notes icon for the day with the first note and get list with the notes for the current month (there should be 3)
17. Open the first note and check the modifications from before
18. Close the Notes list
19. Click on the notes icon for the second day, the list should show two notes (there should be 2)
20. Open one of the notes and delete it
21. Open the second notes and delete it, the notes window should close

Anniversaries

1. Create Good Friday as a holiday (-2 days before Easter Sunday)
2. Create Pentecost as a holiday (49 days after Easter Sunday)
3. Create an anniversary for a date within a period of years, not a holiday
4. Create an anniversary for a date, which is not a holiday
5. Create an anniversary for a date with the current year as the start year
6. Create an anniversary for a date with the current year as the end year
7. Close the list
8. Navigate the calendar and check the anniversaries are shown where they should be
9. Modify the text for the third anniversary
10. Add a start and an end year for the fourth anniversary

11. Add an end year for the fifth anniversary
12. Navigate the calendar again and check the anniversaries are shown correct
13. Open the last anniversary by clicking the anniversary icon and remove it
14. Add an anniversary for the same day as the fifth anniversary
15. Click on the anniversary icon for this day and remove both anniversaries
16. Open the anniversary list by clicking the icon in the toolbar
17. Remove the fourth anniversary
18. Close the list
19. Navigate the calendar and check the anniversaries are shown correct (there should be 3)

The test results in an error. When the program finds anniversaries for a date it does not test for limits on year. This error must be corrected what happens in the class *DateViewModel* and in the method *SetDate()*.

Appointments

1. In the class *MainViewModel* you must set a comment in the two method *ViewAppointments()* and *CheckAppointments()* such these methods does not deletes appointments.
2. Created an appointment for the current day from 10:00 to 11:00
3. Created an appointment for the same day that starts at 12:00 (no end)
4. Created an appointment for the same day from 8:30 to 9:30
5. Created an appointment for the same day, but without time indication. It results in a conflict, and I have agreed to create the appointment anyway
6. Close the program
7. Open the program again
8. Open maintenance appointments from the toolbar
9. Checked that the appointments are sorted correctly
10. Open the second of the above appointments (that without end time) and enter 14:00 as end time. When you save the appointment, you gets a warning about conflicting appointments when the appointment conflicts with the appointment without time indication
11. Open the appointment without indication of time and set the time interval from 15:00 to 16:00.
    When saving, there will be no warning - there is no longer conflicting appointments
12. Created an appointment starting the current day at 17:00 and ends the next day at 9:00
13. Opened the list of appointments, and open the above appointment and deleted it
14. Close the program

15. Open the program again, and opened the list with appointments and check that everything looks correct
    Created an appointment on three different days in the next month - all without time indications
16. Edited the first of these appointments and set a start and an end time, where the last one should be smaller than the first. It should result in an error. Fixed the end time, so it is larger than the start time and saved appointment
17. Edited same appointment again and changed the end date to the day before. It should give an error. The click *Cancel* to cancel the change
18. Close the program and open it again and check that all appointments for the next day are still there
19. Modify the times for all appointments for the current day such all times are after the current time and can so the appointments can be used to check for notifications
20. Close the program an remove the comments from the first test case
21. Run the program again and test the notification.

After test of appointments is found three errors, and the result is then a to-do list:

1. When click on Appointments in the toolbar the list is not sorted correct
2. When updating an appointment the result of *From* an *To* are not updated in the list
3. The buttons in *RemainderWindow* has no action

The next step is then to correct the errors:

1. The SQL statement in the method *GetAppointments()* in the class *DB* is change so ORDER BY also sort on time for the first date.
2. In the class *AppointmentViewModel* are the methods *Add()* and *Update()* changed so the update the properties in the class and not the properties in the model class.
3. In the XML code is added a command element to each of the two buttions.

After these modifications, the test must be performed again.

8.8 WATCH, ALARM AND TIMER

The project copy, which is the copy of the above refactoring, is called *Calendar6*. In this iteration the following functions must be implemented and tested:

1. *Watch*, that show a watch in the window.
2. *Stopwatch*, that implements a stopwatch.
3. *Alarm*, where the user can set an alarm that shows a waring after a given time.
4. *Timer*, where the user can enter a time, and the function shows a warning, when the timer is 0.

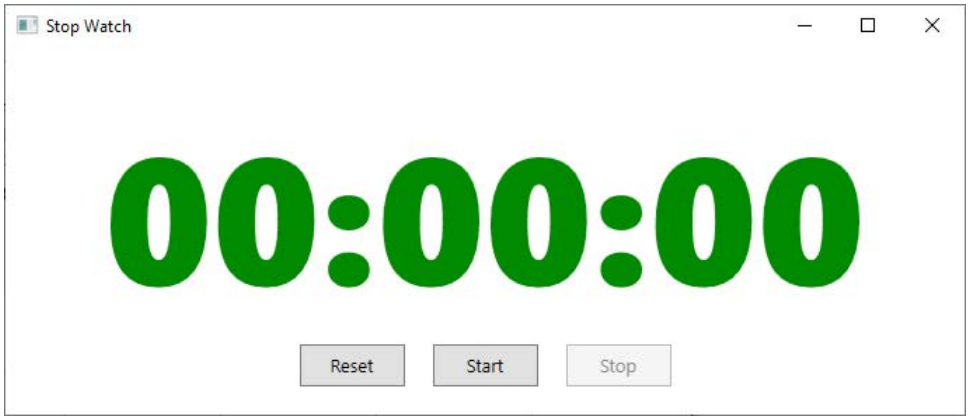
Watch

This is a simple function and should be an on/off function, and the feature shows a watch in the status bar. If you clicks on the icon in the toolbar, the function is turned on and if you clicks again, the function is turned off.

To implement the function the class *MainWindow* starts a thread that ticks every second (an infinity loop, which for each iterations sleeps one second). If the clock is turn on a property is updated with the current time, and a label in the user interface is bound to this property. All changes for this feature concerns only the class *MainViewModel* and the label in *MainWindow*.

Stopwatch

Then there is the stopwatch. It has been decided that it should only be possible to start one stopwatch. If you click on the button in the toolbar (and the stopwatch is not already started), you get the following window:



where you can start the stopwatch. If you do that, an icon appears in the lower right corner of the calendar window, which indicates that the stopwatch is running and if you click this icon, it brings the above dialog box in front, and you can see what the stopwatch displays and possible stop the watch. If you stop the watch, you can start it again, and it will continue from where it was stopped. You can also reset the watch to 0. To close the window and the feature you must click the cross in the tile bar.

The logic for the stop watch is defined in a class *StopViewModel* and is quite simple. The class consists of a property and three commands which all are bound in the user interface. When the user clicks *Start* the command starts a thread which updates the user interface each second, and the only thing to note is, that thread use a dispatcher as it must update a user interface in an other thread.

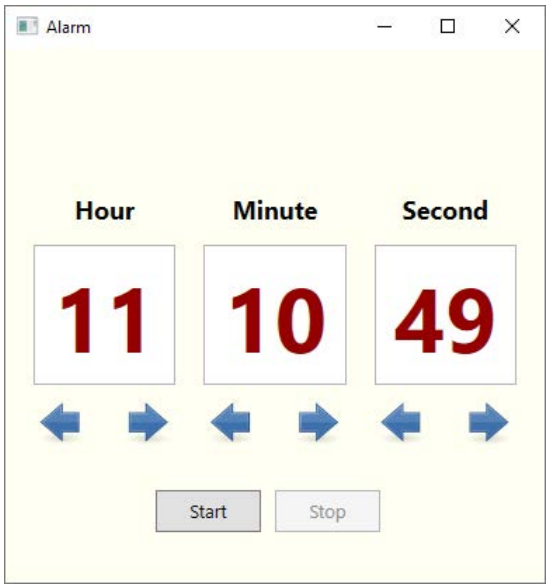
The window is defined as a view class *StopWindow*. It is also a simple class, but you should note how to close the window when clicking the cross. The handler has to notify *MainWindow* as the icon in the lover left corner must disappear.

Alarm and Timer

These two features are basically the same function, so I will mention them in the same place and only show how the feature alarm is implemented. The difference is, that you for an alarm select the clock for where the alarm must occur, and when you start the alarm it each second test if the current time has reached the time for the alarm. A timer starts from 0, and you select the time for when the timer should give an alarm. It's a bit like a minute clock you use in the kitchen.

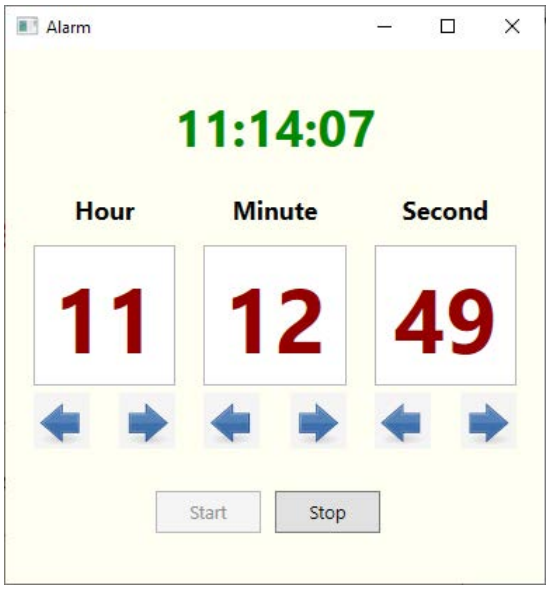
It has been decided that it should only be possible to start one alarm and one timer at a time, and that the state of the two functions should not be persistent. That is, if an alarm is set or a timer is started, and the program is closed, the state of the two functions is not saved and will not restart if the program is reopened.

If you click on the alarm bar in the toolbar, you get the following window:



which shows the current time. You can then set the clock. If you click on start, an icon appears in the lower right corner of the calendar, which tells you that an alarm has been set (in case the dialog should be covered by another window). When the alarm time occurs, you receive a warning as shown below and the alarm function is completed:

Clicking the icon in the lower right corner opens the above dialog again:



The user interface is defined of the class *WatchWindow*, and it is the same class used for both an alarm and a timer. The functionality is defined in the class *WatchViewModel* and the class basically works in the same way as the *StopViewModel* class. However, the class is abstract as the threads must be initialized differently depending on whether it is an alarm or a timer. It means that the class *WatchWindow* instantiates two different models.

8.9 THE LAST FEATURES

There are three functions left:

1. The icons for the moon and the sun
2. Save a calendar as a text file
3. The program settings

The project copy for the start of this iteration is called *Calendar7*.

The program settings

Implementation of this function requires decisions regarding:

1. What settings should be possible
2. Where should the settings be saved

The following settings should be possible:

1. Where should the database be saved
2. The used calendar
3. Longitude and latitude
4. Save appointments on / off
5. Turn notifications for appointments on / off
6. When to notify the user about appointments
7. How often to check for appointments
8. Time out for notifications

The database is just a file and it should be possible to select where to store the database, but also to connect to another database. It must be possible to select the used calendar, and it is the date for the shift from the Julian calendar to the Gregorian calendar. Longitude and latitude and then the location of the machine are used to calculate the position of the sun and the moon. All the other settings are for appointments and are all simple.

The settings must be saved somewhere, and it is decided to save the settings as an XML document in the user's home directory. You can determine this directory with the following statement when I want to save the configuration in the directory *Roaming*:

```
System.Environment.GetFolderPath(Environment.SpecialFolder.UserProfile)
+
  \\AppData\\Roaming\\Calendar
```

When I want to save the configurations as XML it is because an administrator in this way can update the settings if something go wrong.

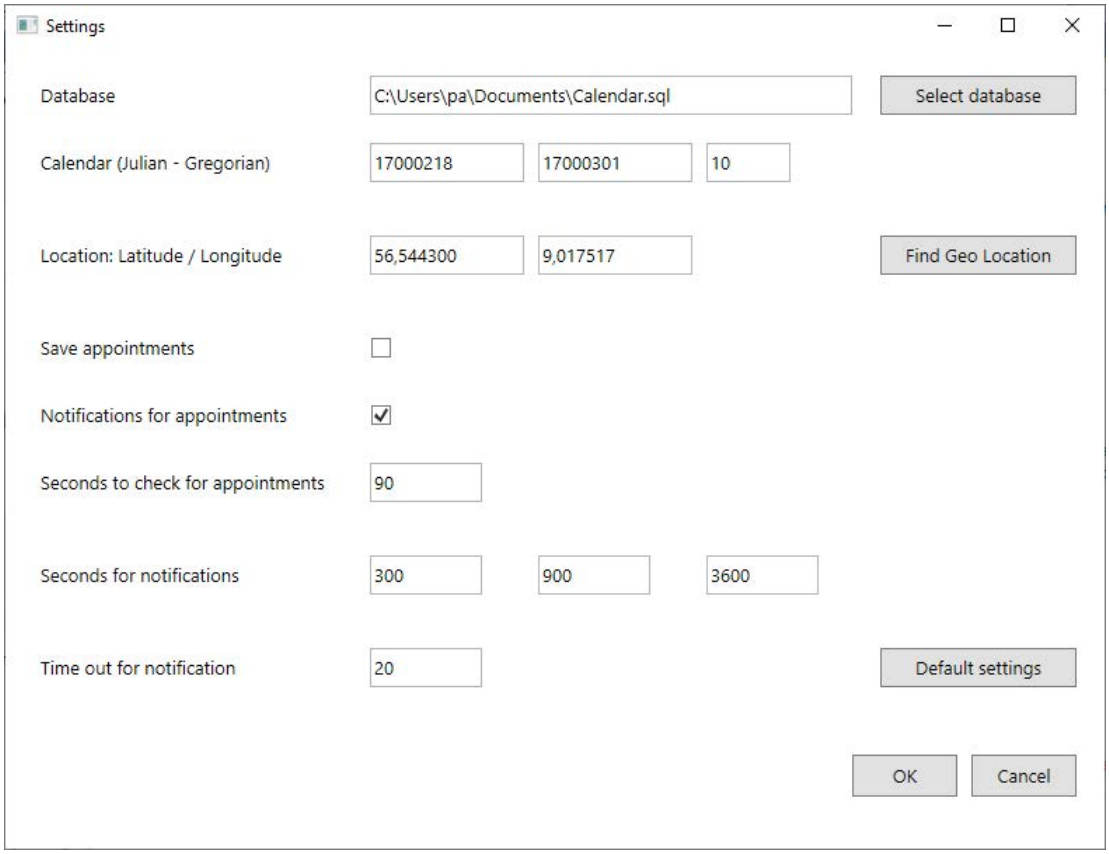
Implementing this functionality is not easy. The model layer is extended with a class *Settings* with properties for the above settings. The class is trivial but is implemented as a singleton. When the program starts this class is initialize and as so the settings are known all over the program.

To load the settings when the program starts and save them when they are changed the data access layer is extended with a class *Config*. This class is not quite simple. The class has two static methods. The first load the settings from the configuration file, but the first time you run the program there are no settings and the program must assign default values.

If there is no value for the database (and the first time you run the program there is no value) the program opens a file browser to let the user select where to save the database and enter the name.

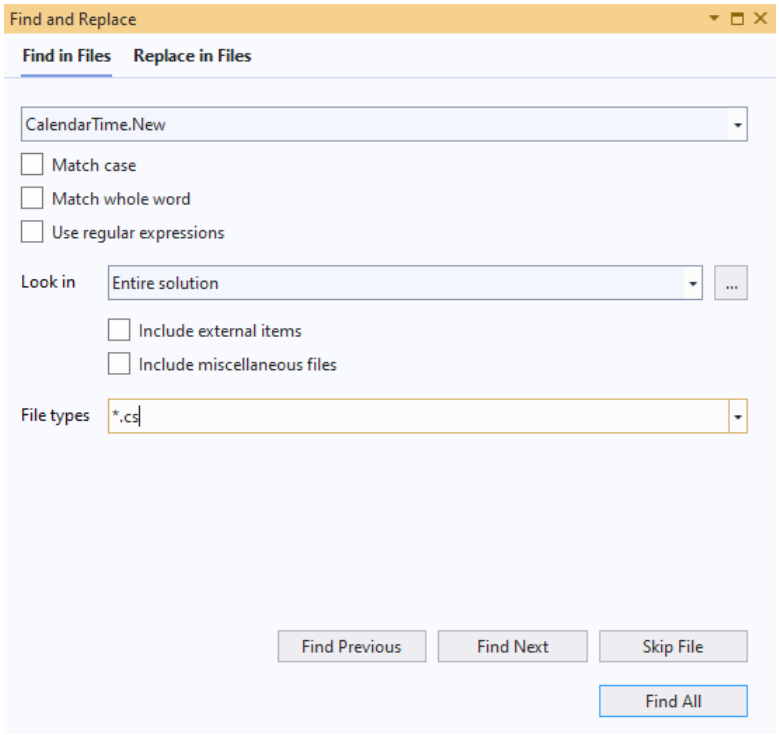
To create a default calendar the program use current culture info, and if it is not the right value the user must later change the value in the settings dialog box. To assign default values for location the program try to read the geo location. If it is not possible (because the computer does not have this facility or it is not turned on) the program opens a dialog box where the user must enter the values. This dialog box is called *GeoWindow* and is place as a class in the data access layer. In principle, this layer should not contain elements from the user interface, but since this dialog box should only be used as part of the program's configuration and is not in fact a part of the program it is located here, and it is also a very simple dialog box.

If you in the tool bar click the icon for settings, the program opens the following window:



There is not much to explain, but the field for the database name is read-only and to change the name you must click the button which opens a file browser. You must note, if you change the name a new database is created if it does not exists. To implement the function is created a view class called *SettingsWindow* and a view model class called *SettingsViewModel* and these two classes does not add something new.

When the function is implemented, and that is maintenance of the singleton *Settings* it must also be used. The class *DB* must use the database name and the code for appointments must use the settings. Also the calendar must be used, and here in fact many places need to be changed. The program already has a class called *CurrentCalendar* (in the model layer). When I delete this class the compiler will find all places where it is used, and I can modify the code to instead use *Settings.Instance.Calendar*. There are, however, a number of other places where *CalendarTime* objects are created and in all places a *Calendar* parameter must also be used. Visual Studio has a search feature that can be used to search these statements:



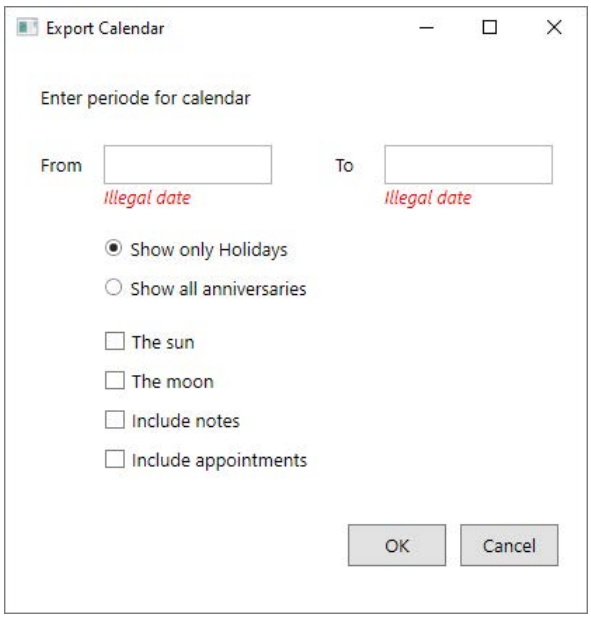
and then add the parameter in all statements where it is not already added.

The moon and the sun

This feature is simple and consists solely of adding two dialog boxes, and for each dialog box the model must instantiate a *Moon* object or a *Sun* object respectively. Otherwise, nothing should happen except that the elements of the dialog box binds to the properties of the objects in question.

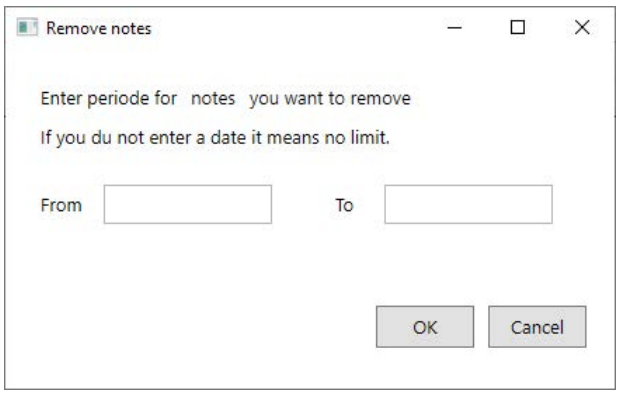
Save a calendar as text

For this feature is a change where to save a calendar as XML instead as a comma separated file. When you select the function from the tool bar you get the below dialog box to select the period for the calendar and which information to be saved. The XML is created in the model for the dialog box, and you should note that if you save a calendar for a big period and include all information the file can become very large.



A last feature

The windows which shows notes and appointments are both expanded with a new button for a function which remove all notes or appointments within a period. If you click the button you get a window to select a period for the items to be removed:



You must note that both fields must be empty, and if so all notes or appointments are deleted.

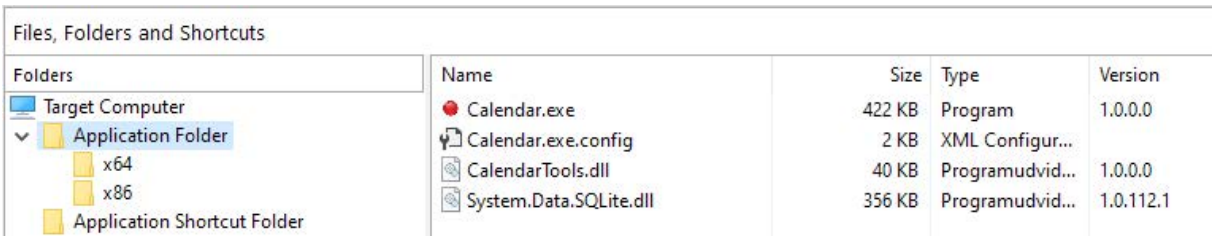
8.10 THE LAST ITERATION

The last iteration also starts with a copy of the project called *Calendar8*. The last step consists of a refactoring and is an iteration where to make some last adjustments and a cleanup in the code. In this case it is the following:

1. An icon is created and assigned the project.
2. To make all windows how the same look-and-feel I have created a few styles in *App.xaml* and used these styles in the programs windows. The styles regarding font and colors.
3. In the most windows I have added default and cancel actions for buttons.
4. For many windows are added an *Activate* event handler.
5. In the folder *Tools* are added three Windows classes which replaces the usual *MessageBox*.

The program has one object of the type *DB*, and when this object must be used in many classes, it is necessary to pass this object round the entire system as a parameter to the individual methods. To avoid that the class *DB* is changed to a singleton and the interface *IDb* is removed. This means that a lot of places need to be changed, but the compiler finds where I needs to change.

As the last step I want to create an installation program using Advanced Installer. First I build a release version of the project. Then I create a setup project to build a MSI file. This time the install package must contain more files, which is the program file and the class library *CalendarTools*, but also the files for SQLite:



When I build the installation file the program can be installed at a machine, and the first time you run the program, a configuration file will be created.