Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

Threads

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Multithreading Konzepte und Anwendung`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/23_Threads.md
Autoren	@author

Motivation - Threads

Bisher haben wir rein sequentiell ablaufende Programme entworfen. Welches Problem generiert dieser Ansatz aber, wenn wir in unserer App einen Update-Service integrieren?

Grundlagen

> Ein Ausführungs-Thread ist die kleinste Sequenz von programmierten Anweisungen, die unabhängig von einem Scheduler verwaltet werden kann, der typischerweise Teil des Betriebssystems ist.

Die Implementierung von Threads und Prozessen unterscheidet sich von Betriebssystem zu Betriebssystem, aber in den meisten Fällen ist ein Thread ein Bestandteil eines Prozesses.

Innerhalb eines Prozesses können mehrere Threads existieren, die gleichzeitig ausgeführt werden und Ressourcen wie Speicher gemeinsam nutzen, während verschiedene Prozesse diese Ressourcen nicht gemeinsam nutzen. Insbesondere teilen sich die Threads eines Prozesses seinen ausführbaren Code und die Werte seiner dynamisch zugewiesenen Variablen und seiner nicht thread-lokalen globalen Variablen zu einem bestimmten Zeitpunkt.

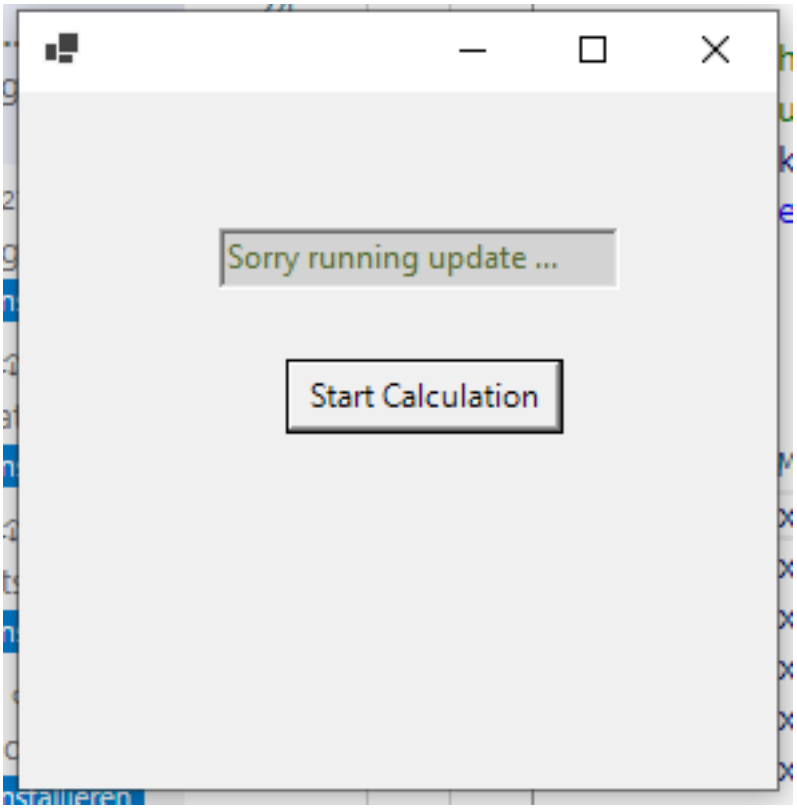
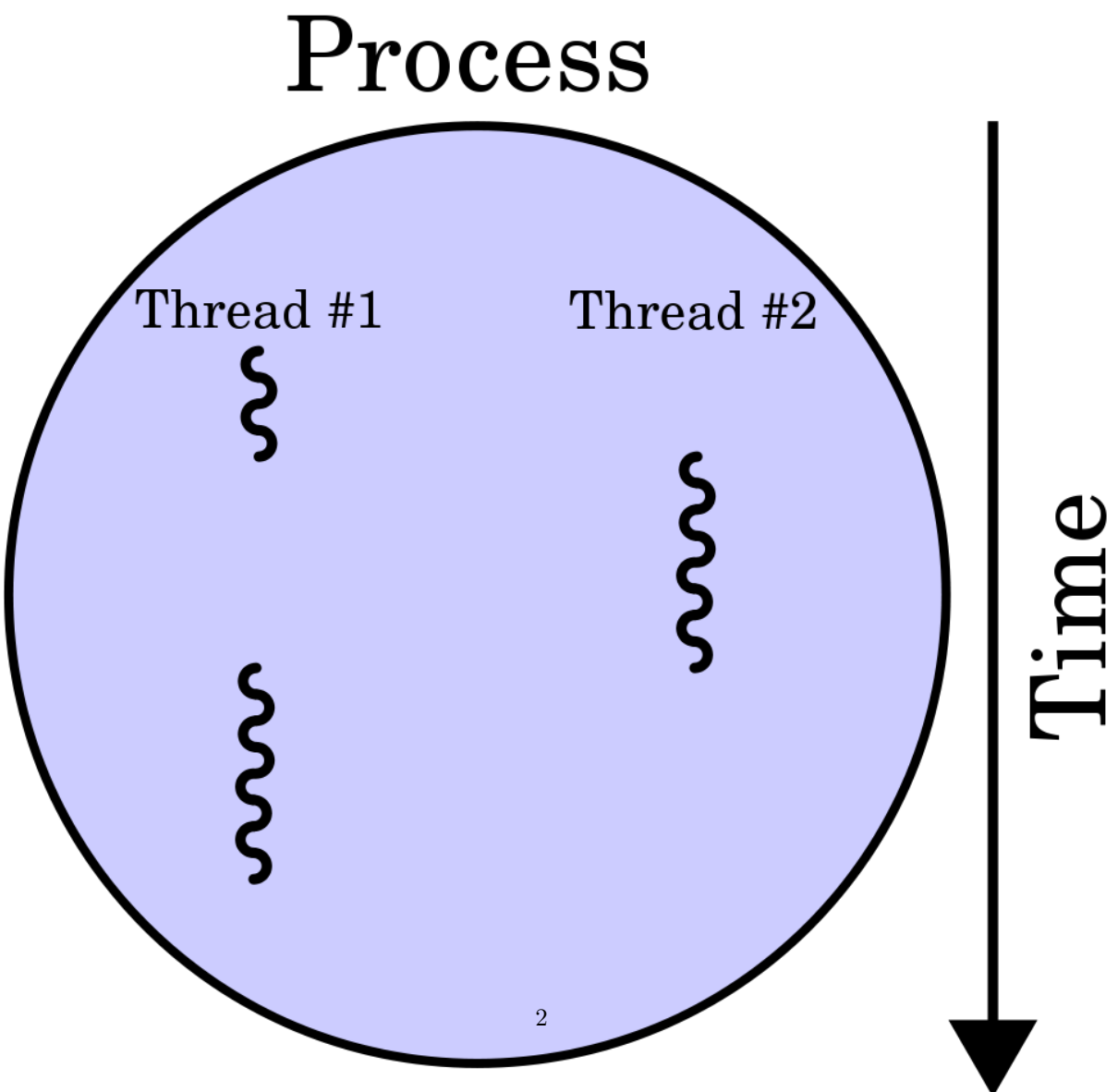


Figure 1: BlockedGUI



Auf eine Single-Core Rechner organisiert das Betriebssystem Zeitscheiben (unter Windows üblicherweise 20ms) um Nebenläufigkeit zu simulieren. Eine Multiprozessor-Maschine kann aber auch direkt auf die Rechenkapazität eines weiteren Prozessors ausweichen und eine echte Parallelisierung umsetzen, die allerdings im Beispiel durch den gemeinsamen Zugriff auf die Konsole limitiert ist.

Vorteile von Multi-Threading Applikationen:

- Ausnutzung der Hardwarefähigkeiten (MultiCore-Systeme) zur Effizienzsteigerung
- Verhinderung eines "Verhungerns" der Anwendung

Erfassung der Performance

Wie messen wir aber die Geschwindigkeit eines Programms?

Implmementierung unter C

Die Implementierung der Klasse Thread unter C# umfasst dabei folgende Definitionen:

```
public delegate void ThreadStart();
public enum ThreadPriority (Normal, AboveNormal, BelowNormal, Highest, Lowest);
public enum ThreadState (Unstarted, Running, Suspended, Stopped, Aborted, ...);

public sealed class Thread{
  public Thread (ThreadStart startMethod);
  ...
  public string Name {get; set;};
  public ThreadPriority Priority {get; set;};
  public ThreadState ThreadState {get;};
  public bool IsAlive {get;};
  public bool IsBackground{get;};
  public void Start();
  public void Join();
  public void Abort(Object);
  public static void Sleep(int milliseconds);
}
```

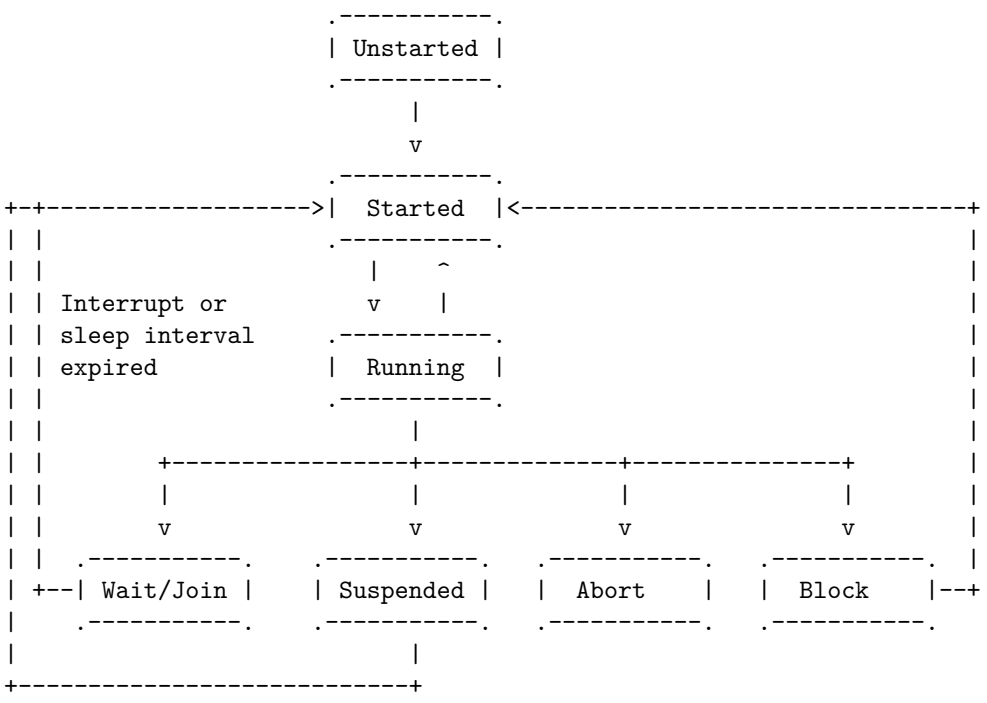
Um die grundlegende Verwendung des Typs `Thread` zu veranschaulichen, nehmen wir an, Sie haben eine Konsolenanwendung, in der die `CurrentThread`-Eigenschaft ein Thread-Objekt abruft, das den aktuell ausgeführten Thread repräsentiert.

```
using System;
using System.Threading;

class Program
{
    public static void Main(string[] args)
    {
        Thread t = Thread.CurrentThread;
        t.Name = "Primary_Thread";
        Console.WriteLine("Thread Name: {0}", t.Name);
        Console.WriteLine("Thread Status: {0}", t.IsAlive);
        Console.WriteLine("Priority: {0}", t.Priority);
        Console.WriteLine("Context ID: {0}", Thread.CurrentContext.ContextID);
        Console.WriteLine("Current application domain: {0}",Thread.GetDomain().FriendlyName);
    }
}
```

```
using System;
using System.Threading;

class Printer{
  char ch;
  int sleepTime;
```

```
  public Printer(char c, int t){
    ch = c;
    sleepTime = t;
  }

  public void Print(){
    for (int i = 0; i<10;  i++){
      Console.Write(ch);
      Thread.Sleep(sleepTime);
    }
  }
}

class Program {
    public static void Main(string[] args){
        Printer a = new Printer ('a', 10);
        Printer b = new Printer ('b', 50);
        Printer c = new Printer ('c', 70);

        var watch = System.Diagnostics.Stopwatch.StartNew();
        a.Print();
        b.Print();
        c.Print();
        watch.Stop();
        Console.WriteLine("\nDuration in ms: {0}", watch.ElapsedMilliseconds);

        watch.Restart();
        Thread PrinterA = new Thread(new ThreadStart(a.Print));
        Thread PrinterB = new Thread(new ThreadStart(b.Print));
        PrinterA.Start();
        PrinterB.Start();
        c.Print();   // Ausführung im Main-Thread
        watch.Stop();
        Console.WriteLine("\nDuration in ms: {0}", watch.ElapsedMilliseconds);
    }
}
```

Thread-Interaktion



`Resume` .

Wie lässt sich eine Serialisierung von Threads realisieren? Im Beispiel soll die Ausführung des Printers C erst starten, wenn die beiden anderen Druckaufträge abgearbeitet wurden.

Methode	Bedeutung
`t.Join()`	Es wird so lange gewartet, bis der Thread t zum Abschluss gekommen ist.
`Thread.Sleep()`	Es wird für n Millisekunden gewartet.
`Thread.Yield()`	Gibt den erteilten Zugriff auf die CPU sofort zurück.

```
using System;
using System.Threading;

class Printer{
  char ch;
  int sleepTime;
  public Printer(char c, int t){
    ch = c;
    sleepTime = t;
  }
  public void Print(){
    for (int i = 0; i<10;  i++){
      Console.Write(ch);
      //Thread.Sleep(sleepTime);
      Thread.Yield();
    }
  }
}
class Program {
    public static void Main(string[] args){
        Printer a = new Printer ('a', 10);
        Printer b = new Printer ('b', 50);
        Printer c = new Printer ('c', 70);
        Thread PrinterA = new Thread(new ThreadStart(a.Print));
        Thread PrinterB = new Thread(new ThreadStart(b.Print));
        PrinterA.Start();
        PrinterB.Start();
        Thread.Sleep(1000);     // Zeitabhängige Verzögerung des Hauptthreads
        //PrinterA.Join();      // <-
        //PrinterB.Join();
        c.Print();
    }
}
```

Aus dem Gesamtkonzept des Threads ergeben sich mehrere Zustände, in denen sich dieser befinden kann:

Zustand	Bedeutung	Transition
unstarted	Thread ist initialisiert	`t.Start();`
running	Thread befindet sich gerade in der Ausführung	
WaitSleepJoin	Thread wird wegen eines Sleep oder eines Join-Befehls nicht ausgeführt. Er nutzt keine Prozessorzeit.	Ablauf des Zeitfensters, Ende des mit `Join()` referenzierten Threads
Suspended	Der Thread ist dauerhaft deaktiviert.	`t.Resume() aktiviert ihn wieder`
stopped	Bearbeitung beendet	

Jeder Thread umfasst eine Feld vom Typ `ThreadState`, dass auf verschiedenen Ebenen dessen Parameter abbildet. Um nur die für uns relevanten Informationen zu erfassen, benutzen wir eine kleine Funktion.

```
public static ThreadState DetermineThreadState(this ThreadState ts){
  return ts & (ThreadState.Unstarted |
               ThreadState.Running |
```

```
                ThreadState.WaitSleepJoin |
                ThreadState.Stopped);

bool blocked = (Thread_a.ThreadState & ThreadState.WaitSleepJoin) != 0;
}
```

Ein Thread in C# zu einem beliebigen Zeitpunkt existiert in einem der folgenden Zustände. Ein Thread liegt zu einem beliebigen Zeitpunkt nur in einem Zustand vor.

Thread-Initialisierung

Wie wird der Thread-Objekt korrekt initialisiert? Viele Tutorials führen Beispiele auf, die wie folgt strukturiert sind, während im obrigen Beispiel der Konstruktoraufruf von `Thread` ein weiteren Konstruktor `ThreadStart` adressiert:

```
Thread threadA = new Thread(ExecuteA);
threadA.Start();
// vs
Thread threadB = new Thread(new ThreadStart(ExecuteB));
```

```
using System;
using System.Threading;

class Calc
{
    int paramA = 0;
    int paramB = 0;

    public Calc(int paramA, int paramB){
      this.paramA = paramA;
      this.paramB = paramB;
    }

    // Static method
    public static void getConst()
    {
        Console.WriteLine("Static funtion const = {0}", 3.14);
    }

    public void process()
    {
        Console.WriteLine("Result = {0}", paramA + paramB);
    }
}

class Program
{
    static void Main()
    {
        ThreadStart threadDelegate = new ThreadStart(Calc.getConst);
        Thread newThread = new Thread(threadDelegate);
        newThread.Start();

        newThread = new Thread(Calc.getConst);    // impliziter Cast zu ThreadStart
        newThread.Start();

        Calc c = new Calc(5, 6);
        threadDelegate = new ThreadStart(c.process);
        newThread = new Thread(threadDelegate);
        newThread.Start();
    }
}
```

Der Konstruktor der Klasse `Thread` hat aber folgende Signatur:

Konstruktor	Initialisiert eine neue Thread Klasse …
`Thread(ThreadStart)`	… auf der Basis einer Instanz von ThreadStart
`Thread(ThreadStart, Int32)`	… auf der Basis eine Instanz von ThreadStart unter Angabe der Größe des Stacks in Byte (aufgerundet auf entsprechende Page Size und unter Berücksichtigung der globalen Mindestgröße)
`Thread(ParameterizedThreadStart)`	… auf der Basis eine Instanz von ParameterizedThreadStart
`Thread(ParameterizedThreadStart, Int32)`	… auf der Basis eine Instanz von ParameterizedThreadStart unter Angabe der Größe des Stacks

```
// impliziter Cast zu ParameterizedThreadStart
Thread threadB = new Thread(ExecuteB);
threadB.Start("abc");

// impliziter Cast und unmittelbarer Start
var threadC new Thread(SomeMethod).Start();
```

Aufgabe: Ergänzen Sie das schon benutzte Beispiel um die Möglichkeit das auszugebene Zeichen als Parameter zu übergeben!

```
using System;
using System.Threading;

class Printer{
  char ch;
  int sleepTime;

  public Printer(char c, int t){
    ch = c;
    sleepTime = t;
  }

  public void Print(int count){
    for (int i = 0; i<count;  i++){
      Console.Write(ch);
      Thread.Sleep(sleepTime);
    }
  }
}

class Program {
    public static void Main(string[] args){
        Printer a = new Printer ('a', 10);
        Thread PrinterA = new Thread(new ThreadStart(a.Print));
        PrinterA.Start();
    }
}
```

Datenaustausch zwischen Threads

Jeder Thread realisiert dabei seinen eigenen Speicher, so dass die lokalen Variablen separat abgelegt werden. Die Verwendung der lokalen Variablen ist entsprechend geschützt.

```
using System;
using System.Threading;

class Program
{
    static void Execute(object output){
      for (int i = 0; i<10;  i++){
        Console.WriteLine(output + i.ToString());
```

```
      }
    }

    public static void Main(string[] args){
        Thread thread_A = new Thread(Execute);
        thread_A.Start("New Tread :");
        Execute("MainTread :");
    }
}
```

Warum werden die beiden Threads ohne Unterbrechung sequentiell abgearbeitet? Welche Ergänzung ist notwendig, um einen zyklischen Wechsel zu erzwingen?

Auf dem individuellen Stack eigenen Kopien der lokale Variablen `count` angelegt, so dass die beiden Threads keine Interaktion realisieren.

Was aber, wenn ein Datenaustausch realisiert werden soll? Eine Möglichkeit der Interaktion sind entsprechende Felder innerhalb einer gemeinsamen Objektinstanz.

Welches Problem ergibt sich aber dabei?

```
using System;
using System.Threading;

class InteractiveThreads
{
  // Gemeinsames Member der Klasse
    //[ThreadStatic] // <- gemeinsames Member innerhalb eines Threads
  public static int count = 0;

  public void AddOne(){
        count++;
    Console.WriteLine("Nachher {0}", count);
  }
}

class Program
{
    public static void Main(string[] args){
        InteractiveThreads myThreads = new InteractiveThreads();
        for (int i = 0; i<100; i++){
          new Thread(myThreads.AddOne).Start();
        }
        Thread.Sleep(10000);
        Console.WriteLine("\n Fertig {0}", InteractiveThreads.count);
    }
}
```

```
using System;
using System.Threading;

class Calc
{
    int paramA = 0;
    public void Inc()
    {
        paramA = paramA + 1;
        Console.WriteLine("Static funtion const = {0}", paramA);
    }
}

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

```
        Calc c = new Calc();
        ThreadStart delThreadA = new ThreadStart(c.Inc);
        Thread newThread_A = new Thread(delThreadA);
        newThread_A.Start();

        ThreadStart delThreadB = new ThreadStart(c.Inc);
        Thread newThread_B = new Thread(delThreadB);
        newThread_B.Start();
    }
}
```

Locking

Locking und Threadsicherheit sind zentrale Herausforderungen bei der Arbeit mit Multithread-Anwendungen. Wie können wir im vorhergehenden Beispiel sicherstellen, dass zwischen dem Laden von threadcount in ein Register, der Inkrementierung und dem zurückschreiben nicht ein anderer Thread den Wert zwischenzeitlich manipuliert hat.

Für eine binäre Variable wird dabei von einem Test-And-Set Mechanisms gesprochen der Thread-sicher sein muss. Wie können wir dies erreichen? Die Prüfung und Manipulation muss atomar ausgeführt werden, dass heißt an dieser Stelle darf der ausführende Thread nicht verdrängt werden.

Darauf aufbauend implementiert C# verschiedene Methoden:

Threadsicherheit	Bemerkung
"exclusive lock"	Alleiniger Zugriff auf eine Codeabschnitt
Monitor	Erweiterter `lock` mit Berücksichtigung von Ausnahmen
Mutex	Prozessübergreifende exklusive Sperrung
Semaphor	Zugriff auf einen Codeabschnitt durch n Threads

```
static readonly object locker = new object();

lock(locker){
  // kritische Region
}
```

```
using System;
using System.Threading;

class InteractiveThreads{
  public int count = 0;
  public void AddOne(){
    lock(this)
    {
       count = count + 1;
       count = count + 1;
       count = count + 1;
       count = count + 1;
    }
    Console.WriteLine("count {0}", count);
  }
}

class Program {
    public static void Main(string[] args){
        InteractiveThreads myThreads = new InteractiveThreads();
        for (int i = 0; i<10; i++){
          new Thread(myThreads.AddOne).Start();
        }
        Thread.Sleep(1000);
    }
}
```

Hintergrund und Vordergrund-Threads

Threads können als Hintergrund- oder Vordergrundthread definiert sein. Hintergrundthreads unterscheiden sich von Vordergrundthreads durch die Beibehaltung der Ausführungsumgebung nach dem Abschluss. Sobald alle Vordergrundthreads in einem verwalteten Prozess (wobei die EXE-Datei eine verwaltete Assembly ist) beendet sind, beendet das System alle Hintergrundthreads.

```
using System;
using System.Threading;

class Printer{
  char ch;
  int sleepTime;

  public Printer(char c, int t){
    ch = c;
    sleepTime = t;
  }

  public void Print(){
    for (int i = 0; i<10;  i++){
      Console.Write(ch);
      Thread.Sleep(sleepTime);
    }
  }
}

class Program {
    public static void printThreadProperties(Thread currentThread){
      Console.WriteLine("{0} - {1} - {2}", currentThread.Name,
                                           currentThread.Priority,
                                           currentThread.IsBackground);
    }

    public static void Main(string[] args){
        Thread MainThread = Thread.CurrentThread;
        MainThread.Name = "MainThread";
        printThreadProperties(MainThread);
        Printer a = new Printer ('a', 170);
        Printer b = new Printer ('b', 50);
        Printer c = new Printer ('c', 10);
        Thread PrinterA = new Thread(new ThreadStart(a.Print));
        PrinterA.IsBackground = false;
        Thread PrinterB = new Thread(new ThreadStart(b.Print));
        printThreadProperties(PrinterA);
        printThreadProperties(PrinterB);
        PrinterA.Start();
        PrinterB.Start();
        c.Print();
    }
}
```

Wie verhält sich das Programm, wenn Sie `Printer_.IsBackground = true;` einfügen?

Ausnahmebehandlung mit Threads

Ab .NET Framework, Version 2.0, erlaubt die CLR bei den meisten Ausnahmefehlern in Threads deren ordnungsgemäße Fortsetzung. Allerdings ist zu beachten, dass die Fehlerbehandlung innerhalb des Threads zu erfolgen hat. Unbehandelte Ausnahmen auf der Thread-Ebene führen in der Regel zum Abbruch des gesamten Programms.

Verschieben Sie die Fehlerbehandlung in den Thread!

```
using System;
using System.Threading;

class Program {
  public static void Calculate(object value){
    Console.WriteLine(5 / (int)value);
  }

  public static void Main(string[] args){
    Thread myThread = new Thread (Calculate);
    try{
      myThread.Start(0);
    }
    catch(DivideByZeroException)
    {
      Console.WriteLine("Achtung - Division durch Null");
    }
  }
}
```

Analog kann das Abbrechen eines Threads als Ausnahme erkannt und in einer Behandlungsroutine organsiert werden.

```
// Beispiel aus Mösenböck, Kompaktkurs C# 7, Seite 159
using System;
using System.Threading;

class Program {
  static void Operate(){
    try{
      try{
        try{
          while (true);
        }catch (ThreadAbortException){Console.WriteLine("inner aborted");}
      }catch (ThreadAbortException){Console.WriteLine("outer aborted");}
    }finally {Console.WriteLine("finally");}
  }

  public static void Main(string[] args){
    Thread myThread = new Thread (Operate);
    myThread.Start();
    Thread.Sleep(1);
    myThread.Abort();   // <- Expliziter Abbruch des Threads
    myThread.Join();
    Console.WriteLine("done");
  }
}
```

Thread-Pool

Wann immer ein neuer Thread gestartet wird, bedarf es einiger 100 Millisekunden, um Speicher anzufordern, ihn zu initialisieren, usw. Diese relativ aufwändige Verfahren wird durch die Nutzung von ThreadPools beschränkt, da diese als wiederverwendbare Threads vorgesehen sind.

Die `System.Threading.ThreadPool`-Klasse stellt einer Anwendung einen Pool von "Arbeitsthreads" bereit, die vom System verwaltet werden und Ihnen die Möglichkeit bieten, sich mehr auf Anwendungsaufgaben als auf die Threadverwaltung zu konzentrieren.

```
using System;
using System.Threading;

class Program {
  // This thread procedure performs the task.
```

```
  static void Operate(Object stateInfo)
  {
      Console.WriteLine("Hello from the thread pool.");
  }

  public static void Main(string[] args){
    ThreadPool.QueueUserWorkItem(Operate);
    Console.WriteLine("Main thread does some work, then sleeps.");
    Thread.Sleep(1000);
    Console.WriteLine("Main thread exits.");
  }
}
```

Das klingt sehr praktisch, was aber sind die Einschränkungen?

- Für die Threads können keine Namen vergeben werden, damit wird das Debugging ggf. schwieriger.
- Pooled Threads sind immer Background-Threads
- Sie können keine individuellen Prioritäten festlegen.
- Blockierte Threads im Pool senken die entsprechende Performance des Pools

Aufgaben der Woche

- [ ]