

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 |
| | | fb89zila | | DEVensiv |

Tasks

| Parameter | Kursinformationen |
|-----------------------|---|
| Veranstaltung: | Vorlesung Softwareentwicklung |
| Semester: | Sommersemester 2021 |
| Hochschule: | Technische Universität Freiberg |
| Inhalte: | Tasks und deren Anwendung |
| Link auf den | https://github.com/TUBAF-IfL-LiaScript/VL_Softwareentwicklung/blob/master/24_Tasks.md |
| GitHub: | |
| Autoren | @author |

Logging

Unsere Ausgaben in der Console können auf Dauer ziemlich nerven ...

Lösung: Verwenden Sie ein Logging Framework!

- <https://logging.apache.org/log4net/release/manual/configuration.html>
- <https://blog.elmah.io/log4net-tutorial-the-complete-guide-for-beginners-and-pros/>

```
<log4net>
  <!-- A1 is set to be a ConsoleAppender -->
  <appender name="A1" type="log4net.Appender.ConsoleAppender">

    <!-- A1 uses PatternLayout -->
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread] %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <!-- Set root logger level to DEBUG and its only appender to A1 -->
  <root>
    <level value="DEBUG" />
    <appender-ref ref="A1" />
  </root>
</log4net>
```

Tasks

Die prozedurale/objektorientierte Programmierung basiert auf der Idee, dass ausgehend von einem Hauptprogramm Methoden aufgerufen werden, deren Abarbeitung realisiert wird und danach zum Hauptprogramm zurückgekehrt wird.

```

using System;
using System.Threading;

class Program
{
    public static void TransmitsMessage(string output){
        Random rnd = new Random();
        Thread.Sleep(1);
        Console.WriteLine(output);
    }

    public static void Main(string[] args){
        TransmitsMessage("Here we are");
        TransmitsMessage("Best wishes from Freiberg");
        TransmitsMessage("Nice to meet you");
    }
}

```

An dieser Stelle spricht man von **synchronen** Methodenaufrufen. Das Hauptprogramm (Rufer oder Caller) stoppt, wartet auf den Abschluss des aufgerufenen Programms und setzt seine Bearbeitung erst dann fort. Das blockierende Verhalten des Rufers generiert aber einen entscheidenden Nachteil - eine fehlende Reaktionsfähigkeit für die Zeit, in der die aufgerufene Methode zum Beispiel eine Netzwerkverbindung aufbaut, Daten speichert oder Berechnungen realisiert.

Der Rufer könnte in dieser Zeit auch andere Arbeiten umsetzen. Dafür muss er aber nach dem Methodenaufruf die Kontrolle zurück bekommen und kann dann weiterarbeiten.

Ein Beispiel aus der "Praxis" - Vorbereitung eines Frühstücks:

1. Schenken Sie sich eine Tasse Kaffee ein.
2. Erhitzen Sie eine Pfanne, und braten Sie darin zwei Eier.
3. Braten Sie drei Scheiben Frühstücksspeck.
4. Toasten Sie zwei Scheiben Brot.
5. Bestreichen Sie das getoastete Brot mit Butter und Marmelade.
6. Schenken Sie sich ein Glas Orangensaft ein.

Das anschauliche Beispiel entstammt der Microsoft Dokumentation und ist unter <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/async/> zu finden.

Eine Lösung für diesen Ansatz könnten Threads bieten.

```

using System;
using System.Threading;

class Program {
    static public int[] Result = { 0, 0, 0};
    static Random rnd = new Random();

    public static void TransmitsMessage(object index){
        Console.WriteLine("Thread {0} started!", Thread.CurrentThread.ManagedThreadId);
        // doing some fancy things here
        int delay = rnd.Next(200, 500);
        Thread.Sleep(delay); // arbitrary duration
        Result[(int)index]= delay;
        Console.WriteLine("\nThread {0} says Hello", Thread.CurrentThread.ManagedThreadId);
    }

    public static void Main(string[] args){
        Thread ThreadA = new Thread (TransmitsMessage);
        ThreadA.Start(0);
        Thread ThreadB = new Thread (TransmitsMessage);
        ThreadB.Start(1);
        Thread ThreadC = new Thread (TransmitsMessage);
        ThreadC.Start(2);
    }
}

```

```

    for (int i = 0; i<50; i++){
        Console.Write("*");
        Thread.Sleep(1);
    }
    Console.WriteLine();
    Console.WriteLine("Well done, so far!");
    ThreadA.Join();
    ThreadB.Join();
    ThreadC.Join();
    Console.WriteLine("Aus die Maus!");
    foreach(int i in Result){
        Console.Write("{0} ", i);
    }
}
}

```

Welche Nachteile sehen Sie in dieser Lösung?

Task Modell in C

C# stellt für die asynchrone Programmierung einen neuen Typen **Task** zur Verfügung und für die **await** und **async** Keywörter ein.

Die **Task**-Klasse bildet einen einzelnen Vorgang ab, gibt keinen Wert zurück und wird in der Regel asynchron ausgeführt. **Task** Objekte sind eine der zentralen Komponenten von der aufgabenbasierte asynchrone Muster in .NET Framework 4 eingeführt wurden. Da die Arbeit, indem geleistet eine **Task** Objekt in der Regel führt asynchron auf einem Threadpool-Thread anstatt synchron auf dem Hauptanwendungsthread, Sie verwenden die Thread Status-Eigenschaften des Threads, als auch die **IsCanceled**, **IsCompleted**, und **IsFaulted** Eigenschaften, um den Status eines Vorgangs zu bestimmen. In den meisten Fällen wird ein Lambda-Ausdruck verwendet, um die eigentliche Aufgabe zu spezifizieren.

```

public class Task{
    public Task (Action a);
    public TaskStatus Status {get;}
    public bool IsCompleted {get;}
    public static Task Run(Action a);
    public static Task Delay(int n);
    public void Wait();
    ...
}

public class Task<T>: Task{
    public Task (Func<T> f);
    ...
    public static Task<T> Run (Func <T> f);
    ...
}

```

Die Anwendung erfolgt dabei dem Muster:

```

Task task = new Task(() => {... Anweisungsblock ...});
Task.Start();

```

Hierbei wird deutlich, dass das **Task**-Objekt auf einem **Thread** aufbaut und lediglich eine höhere Abstraktionsstufe darstellt. Der verkürzte Aufruf mittels der statischen **Run**-Methode realisiert das gleiche Verhalten:

```

Task task = Task.Run(() => {... Anweisungsblock ...});

```

Es wäre nun möglich diesen laufenden **Task** aus dem Main-Thread anhand seiner Variablen **IsCompleted** zu überwachen oder mit **join** erfassen, um die Fertigstellung zu erkennen. Dieses Verhalten lässt sich zwar auch mit Threads umsetzen, mit dem höheren Abstraktionsgrad lässt sich die Komplexität des Aufrufes aber reduzieren.

Um für die Durchführung einer einzelnen Aufgabe zu warten, rufen Sie die **Task.Wait** Methode. Ein Aufruf der **Wait** Methode blockiert den aufrufenden Thread, bis die Instanz der Klasse die Ausführung abgeschlossen hat.

```

// Motiviert aus
// https://docs.microsoft.com/de-de/dotnet/api/system.threading.tasks.task?view=netframework-4.8
using System;
using System.Threading.Tasks;
using System.Threading;

public class Example
{
    public static void doSomething(){
        Console.WriteLine("Say hello!");
    }

    public static void Main()
    {
        Action<object> action = (object obj) =>
        {
            Console.WriteLine("Task={0}, obj={1}, Thread={2}",
                Task.CurrentId, obj,
                Thread.CurrentThread.ManagedThreadId);
        };

        // Create a task but do not start it.
        Task t1 = new Task(action, "alpha");
        t1.Start();
        Console.WriteLine("t1 has been launched. (Main Thread={0})",
            Thread.CurrentThread.ManagedThreadId);

        // Nur der Vollständigkeit halber ...
        Task t2 = new Task(doSomething);
        t2.Start();
        Console.WriteLine("t2 has been launched. (Main Thread={0})",
            Thread.CurrentThread.ManagedThreadId);

        Task t3 = Task.Run( () => {
            // Just loop.
            int ctr = 0;
            for (ctr = 0; ctr <= 1000000; ctr++)
            {}
            Console.WriteLine("Finished {0} loop iterations",
                ctr);
        } );

        t3.Wait();
    }
}

```

Wait ermöglicht auch die Beschränkung der Wartezeit auf ein bestimmtes Zeitintervall. Die `Wait(Int32)` und `Wait(TimeSpan)` Methoden blockiert den aufrufenden Thread, bis die Aufgabe abgeschlossen ist oder ein Timeoutintervall abläuft, welcher Fall zuerst eintritt.

```

using System;
using System.Threading;
using System.Threading.Tasks;

class Program {
    public static void Main(string[] args){
        // Wait on a single task with a timeout specified.
        Task taskA = Task.Run( () => Thread.Sleep(2000));
        try {
            taskA.Wait(1000);           // Wait for 1 second.
            bool completed = taskA.IsCompleted;
            Console.WriteLine("Task A completed: {0}, Status: {1}",

```

```

        completed, taskA.Status);
    if (! completed)
        Console.WriteLine("Timed out before task A completed.");
}
catch (AggregateException) {
    Console.WriteLine("Exception in taskA.");
}
}
}

```

Für komplexe Taskstrukturen kann man diese zum Beispiel in Arrays arrangieren. Für diese Reihe von Aufgaben jeweils durch Aufrufen der `Wait` Methode zu warten wäre aufwändig und wenig praktisch. `WaitAll` schließt diese Lücke und erlaubt eine übergreifende Überwachung.

Das folgenden Beispiel werden zehn Aufgaben erstellt, die wartet, bis alle zehn abgeschlossen werden, und klicken Sie dann ihren Status angezeigt.

```

using System;
using System.Threading;
using System.Threading.Tasks;

class Program {
    public static void Main(string[] args){
        // Wait for all tasks to complete.
        Task[] tasks = new Task[10];
        for (int i = 0; i < 10; i++)
        {
            tasks[i] = Task.Run(() => Thread.Sleep(2000));
        }

        try {
            Task.WaitAll(tasks);
        }
        catch (AggregateException ae) {
            Console.WriteLine("One or more exceptions occurred: ");
            foreach (var ex in ae.Flatten().InnerExceptions)
                Console.WriteLine("    {0}", ex.Message);
        }
        Console.WriteLine("Status of completed tasks:");
        foreach (var t in tasks)
            Console.WriteLine("    Task #{0}: {1}", t.Id, t.Status);
    }
}

```

Der Kanon der Möglichkeiten wird aber deutlich erweitert, wenn ein konkreter Rückgabewert genutzt werden soll. Anstatt wie bei Threads mit einer entsprechenden “außen stehenden” Variablen zu arbeiten, wird das Ergebnis im `Task`-Objekt selbst gespeichert und kann dann abgerufen werden. Dieser Aspekt wird über die generische Konfiguration des `Tasks` abgebildet:

```

Task<int> task = Task.Run(() => {int i;
                                //... Anweisungsblock ...;
                                return i});

```

Wie ist dieser Aufruf zu verstehen? Unser `Task` gibt anders als bei der synchronen Abarbeitung nicht unmittelbar mit dem Ende der Bearbeitung einen Wert zurück, sondern verspricht zu einem späteren Zeitpunkt einen Wert in einem bestimmten Format zu liefern. Dank der generischen Realisierung können dies beliebige Objekte sein.

Wie aber erfolgt die Rückgabe wann?

Asynchrone Methoden

Eine asynchrone Methode ruft einen `Task` auf, setzt die eigene Bearbeitung aber fort und wartet auf dessen Beendigung.

```

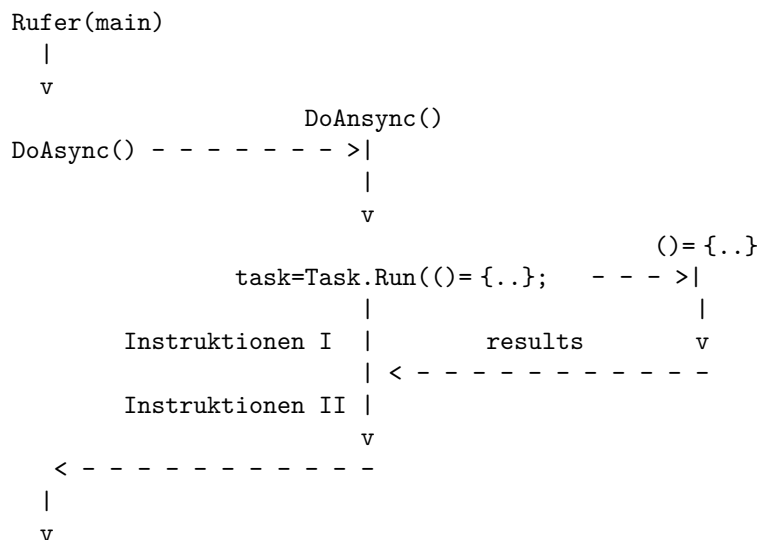
aync void DoAsync(){
    Task<int> task = Task.Run(() => {int i;
                                   // Berechnungen
                                   return i;})

    // Instruktionen I
    // Methoden, die nach der Rückkehr nach DoAsync ausgeführt werden.
    int result = await task;
    // Instruktionen II
    // Hier wird nun mit dem Ergebnis result weitergearbeitet
}

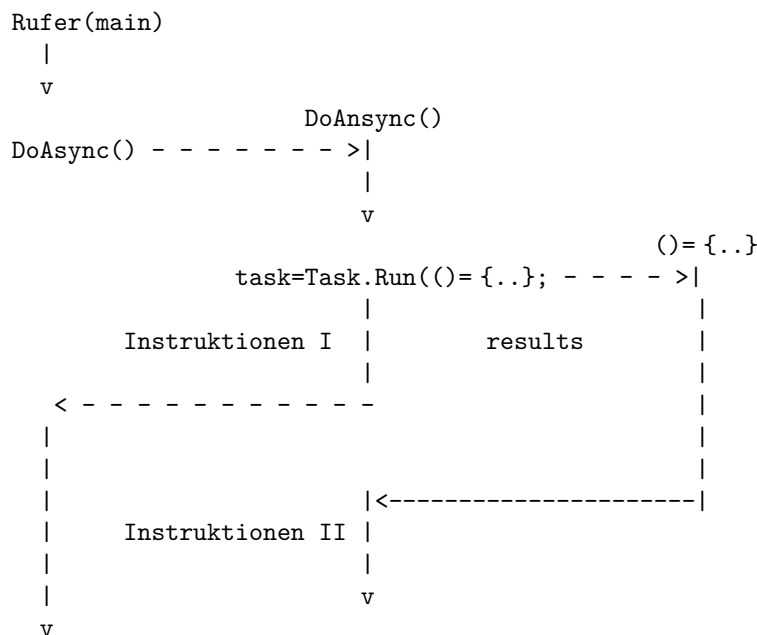
```

Das Ergebnis der Operation hängt dabei davon ab, welche Zeitabläufe sich im Programmablauf ergeben.

Fall I Das Ergebnis der Lambdafunktion liegt vor, bevor DoAsync die Zeile mit await erreicht hat (Quasi-Synchroner Fall)



Fall II Das Ergebnis der Lambdafunktion liegt erst später, nachdem DoAsync die Zeile mit await erreicht hat (und bereits nach main zurückgekehrt ist)



Zwei sehr anschauliche Beispiele finden sich im Code Ordner des Projekts.

| Beispiel | Bemerkung |
|----------------------------------|--|
| AsyncExampleI.cs | Generelle Einbettung des asynchronen Tasks |

| Beispiel | Bemerkung |
|-----------------------------------|--|
| AsyncExampleII.cs | Illustration der Interaktionsfähigkeit eines asynchronen Programmes, das Berechnungen und Nutzereingaben gleichermaßen realisiert. |

Aufgaben der Woche

- []