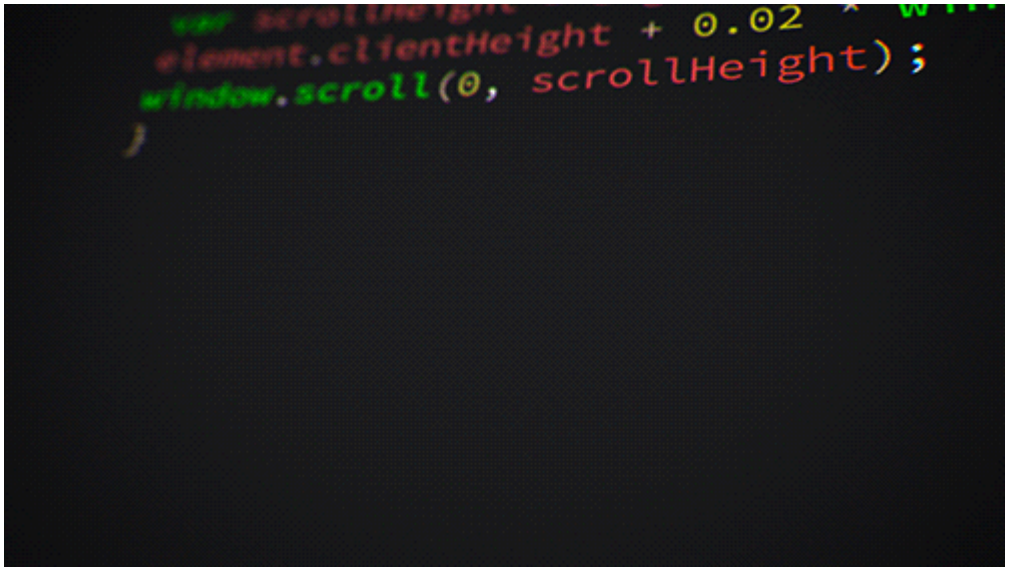
Tasks

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Teil:	`24/27`
Semester	`Sommersemester 2025`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	Logging in Software, Konfiguration eines Programmweiten Loggingsystems, weiterführenden Abstraktionen für Multithreading, Task Modell in C#, asynchrone Methoden
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/24_Tasks.md
Autoren	Sebastian Zug, Galina Rudolf & André Dietrich



Exkurs - Paketmanagement

> **Merke:** Erfinde das Rad nicht neu!

Wie schaffen es erfahrene Entwickler innerhalb kürzester Zeit Prototypen mit beeindruckender Funktionalität zu entwerfen? Sicher, die Erfahrung spielt hier eine große Rolle aber auch die Wiederverwendung von existierendem Code. Häufig wiederkehrende Aufgaben wie zum Beispiel:

- das Logging
- der Zugriff auf Datenquellen
- mathematische Operationen
- Datenkapselung und Abstraktion
- ...

werden bereits durch umfangreiche Bibliotheken implementiert und werden entsprechend nicht neu geschrieben.

Ok, dann ziehe ich mir eben die zugehörigen Repositories in mein Projekt und kann die Bibliotheken nutzen. In individuell genutzten Implementierungen mag das ein gangbarer Weg sein, aber das Wissen um die zugehörigen Abhängigkeiten - Welche Subbibliotheken und welches .NET Framework werden vorausgesetzt? - liegt so nur implizit vor.

Entsprechend brauchen wir ein Tool, mit dem wir die Abhängigkeiten UND den eigentlichen Code kombinieren und einem Projekt hinzufügen können. `NuGet` löst diese Aufgabe für .NET und schließt auch gleich die Mechanismen zur Freigabe von Code ein. NuGet definiert dabei, wie Pakete für .NET erstellt, gehostet und verarbeitet werden.

Ein `NuGet`-Paket ist eine gepackte Datei mit der Erweiterung `.nupkg` die:

- den kompilierten Code (DLLs),
- ein beschreibendes Manifest, in dem Informationen wie die Versionsnummer des Pakets, ggf. der Speicherort des Source Codes oder die Projektwebseite enthalten sind sowie
- die Abhängigkeiten von anderen Paketen und dessen Versionen

enthalten sind Ein Entwickler, der seinen Code veröffentlichen möchte generiert die zugehörige Struktur und läd diese auf einen `NuGet` Server. Unter dem Link kann dieser durchsucht werden.

Anwendungsbeispiel: Symbolisches Lösen von Mathematischen Gleichungen

Eine entsprechende Bibliothek steht unter Projektwebseite. Das Ganze wird als `Nuget` Paket gehostet MathNet.

Unter der Annahme, dass wir `dotnet` als Buildtool benutzen ist die Einbindung denkbar einfach.

```
dotnet new console -o SymbolicMath
cd SymbolicMath
dotnet add package MathNet.Symbolics
Determining projects to restore...
Writing /tmp/tmpNsaYtc.tmp
info : Adding PackageReference for package 'MathNet.Symbolics' into project
  /home/zug/Desktop/Vorlesungen/VL_Softwareentwicklung/code/16_Testen
  /ConditionalBuild/ConditionalBuild.csproj'
```

```
  /conditionalBuild/conditionalBuild.csproj ...
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/mathnet.symb
  /index.json
...
```

Danach findet sich in unserer Projektdatei `.csproj` ein entsprechender Eintrag

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="MathNet.Symbolics" Version="0.24.0" />
  </ItemGroup>
</Project>
```

PreprocessorConsts.cs

```
1  using System;
2  using System.Collections.Generic;
3  using MathNet.Symbolics;
4  using Expr = MathNet.Symbolics.SymbolicExpression;  // Platzhalter fü
     verkürzte Schreibweise
5
6  class Program
7  {
8    static void Main(string[] args)
9    {
10     Console.WriteLine("Beispiele für die Verwendung des MathNet.Symbo
         Paketes");
11     var x = Expr.Variable("x");
12     var y = Expr.Variable("y");
13     var a = Expr.Variable("a");
14     var b = Expr.Variable("b");
15     var c = Expr.Variable("c");
16     var d = Expr.Variable("d");
17     Console.WriteLine("a+a+a =" + (a + a + a).ToString());
18     Console.WriteLine("(2 + 1 / x - 1) =" + (2 + 1 / x - 1).ToString(
19     Console.WriteLine("((a / b / (c * a)) * (c * d / a) / d) =" + ((a
         (c * a)) * (c * d / a) / d).ToString());
20     Console.WriteLine("Der zugehörige Latex Code lautet " + ((a / b /
         a)) * (c * d / a) / d).ToLaTeX());
21   }
22 }
```

xml PreprocessorConsts.csproj

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="MathNet.Symbolics" Version="0.24.0" />
  </ItemGroup>
</Project>
```

```
Beispiele für die Verwendung des MathNet.Symbolics Paketes
a+a+a =3*a
(2 + 1 / x - 1) =1 + 1/x
((a / b / (c * a)) * (c * d / a) / d) =1/(a*b)
Der zugehörige Latex Code lautet \frac{1}{ab}
Beispiele für die Verwendung des MathNet.Symbolics Paketes
a+a+a =3*a
(2 + 1 / x - 1) =1 + 1/x
((a / b / (c * a)) * (c * d / a) / d) =1/(a*b)
Der zugehörige Latex Code lautet \frac{1}{ab}
```

Exkurse: Logging

Wie arbeiten wir bisher in Bezug auf Textausgaben?

ImplicitConstructorCall

```
1  using System;
2  using System.Reflection;
3  using System.ComponentModel.Design;
4
5  public class Person {
6    public int geburtsjahr;
7    public string name;
8
9    public Person(){
10     geburtsjahr = 1984;
11     name = "Orwell";
12     Console.WriteLine("ctor of Person");
13   }
14
15   public Person(int auswahl){
16     if (auswahl == 1) {name = "Micky Maus";}
17     else {name = "Donald Duck";}
18   }
19 }
20
21 public class Fußballspieler : Person {
22   public byte rückennummer;
23 }
24
25 public class Program
26 {
27   public static void Main(string[] args){
28     Fußballspieler champ = new Fußballspieler();
29     Console.WriteLine("{0,4} - {1}", champ.geburtsjahr, champ.name );
30   }
31 }
```

```
ctor of Person
1984 - Orwell
ctor of Person
1984 - Orwell
```

Dieses Vorgehen kann auf Dauer ziemlich nerven ...

Lösung: Verwenden Sie ein Logging Framework, z.B. NLog - ein Logging-Framework für .NET-Anwendungen!

Merkmal	Beschreibung	`print()`	Logging-Framework
Zentrale Steuerung	Konfiguration und Steuerung der Ausgabe zentral möglich	❌	✅
Log-Level	Nachrichten können je nach Wichtigkeit kategorisiert werden	❌	✅
Formatierung	Ausgaben können standardisiert formatiert werden (z. B. mit Zeitstempel)	❌	✅
Dateihandling	Logs können automatisch in Dateien geschrieben und rotiert werden	❌	✅
Mehrere Ausgaben	Gleichzeitige Ausgabe an Konsole, Datei, Netzwerk usw.	❌	✅
Thread-Sicherheit	Gleichzeitige Ausgaben mehrerer Threads führen nicht zu vermischten Zeilen	❌	✅
Integration	Logs können mit externen Tools (z. B. Logserver, Dashboards) verwendet werden	❌	✅

NLog:

- ermöglicht das Protokollieren von Informationen, Warnungen, Fehlern und anderen Ereignissen,
- unterstützt Datei-Logging, Datenbank-Logging, E-Mail-Logging, Konsolen-Logging und mehr

nlog.config:

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <targets>
    <target name="logfile" xsi:type="File" fileName="file.txt" />
```

```
        <target name="logconsole" xsi:type="Console" />
    </targets>

    <rules>
        <logger name="*" minlevel="Info" writeTo="logconsole" />
        <logger name="*" minlevel="Debug" writeTo="logfile" />
    </rules>
</nlog>
```

```
using NLog;

public class Program
{
    private static Logger logger = LogManager.GetCurrentClassLogger();

    public static void Main()
    {
        logger.Info("Anwendung gestartet");
        // ... Weitere Anwendungslogik ...
        logger.Error("Ein Fehler ist aufgetreten");
        // ... Weitere Anwendungslogik ...
        logger.Info("Anwendung beendet");
    }
}
```

- https://github.com/NLog
- https://riptutorial.com/nlog

Rückblick: Multithreading

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.

SynchronOperation.cs

```
1  using System;
2  using System.Threading;
3
4  class Program
5  {
6    public static void TransmitsMessage(string output){
7      Thread.Sleep(1);
8      Console.WriteLine(output);
9    }
10
11   public static void Main(string[] args){
12     TransmitsMessage("Here we are");
13     TransmitsMessage("Best wishes from Freiberg");
14     TransmitsMessage("Nice to meet you");
15   }
16 }
```

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.

Program.cs

```
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);
    // int delay = Random.Shared.Next(200, 500);
    //static ThreadLocal<Random> rnd = new ThreadLocal<Random>(() =>
      Random());
    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);
    }
  }
}
```

xml AsynchonousBehaviour.csproj

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="MathNet.Symbolics" Version="0.24.0" />
  </ItemGroup>
</Project>
```

Welche Nachteile sehen Sie in dieser Lösung?

Monitor.cs

```
using System;
using System.Threading;

public class Fass
{
    private int max, level;
    private readonly object lockObject = new object();

    public Fass(int max)
    {
        this.max = max;
        this.level = 0;
    }

    public void Füllen(int x)
    {
        while (true){
            lock (lockObject)
            {
                while ((level + x) > max)
                {
                    Monitor.Wait(lockObject);
                }
                level += x;
                Console.WriteLine("plus " + x + " level " + level);
                Monitor.PulseAll(lockObject);
            }
            Thread.Sleep(500);
        }
    }

    public void Leeren(int x)
    {
        while (true){
            lock (lockObject)
            {
                while ((level - x) < 0)
                {
                    Monitor.Wait(lockObject);
                }
                level -= x;
                Console.WriteLine("minus " + x + " level " + level);
                Monitor.PulseAll(lockObject);

            }
            Thread.Sleep(500);
        }
    }
```

```
49  }
50
51  class Program
52  {
53      static void Main()
54      {
55          Fass fass = new Fass(500);
56          new Thread(() => fass.Füllen(30)).Start();
57          new Thread(() => fass.Leeren(10)).Start();
58      }
59  }
```

```
plus 30 level 30
minus 10 level 20
```

Task Modell in C#

C# stellt für die asynchrone Programmierung die neuen Typen `Task` und `Task<TResult>` und die Schlüsselwörter `await` und `async` zur Verfügung. Diese sind zentrale Komponenten von den aufgabenbasierten asynchronen Muster (TAP - Task based Asynchronous Pattern), die in .NET Framework 4 eingeführt wurden.

Aspekt	`Thread`	`Task`
Zweck	Repräsentiert einen physischen Ausführungspfad (Thread of Execution).	Repräsentiert eine asynchrone, geplante Arbeitseinheit auf höherer Abstraktionsebene.
Erstellung	Mit `new Thread()` explizit erstellt.	Mit `Task.Run()` oder `Task.Factory.StartNew()` gestartet, meist über Thread-Pool.
Verwaltung	Muss manuell gestartet und verwaltet werden.	Wird vom .NET-Thread-Pool verwaltet.
Rückgabewert	Kein direkter Rückgabewert. Ergebnisse müssen über gemeinsame Variablen oder Callbacks verarbeitet werden.	Kann ein Ergebnis mit `Task<TResult>` zurückgeben.
Stornierung	Keine native Unterstützung. Muss manuell implementiert werden.	Unterstützt Abbruch über `CancellationToken`.
Async-Unterstützung	Nicht für `async`/`await` geeignet.	Vollständig integrierbar mit `async`/`await`.
Ressourcenverbrauch	Teurer, da jeder Thread eigene Ressourcen verwendet.	Ressourcenschonender durch Wiederverwendung im Thread-Pool.
Nebenläufigkeit	Führt exakt einen Codepfad aus.	Kann beliebig viele Tasks gleichzeitig verwalten (abhängig von Systemressourcen).
Abstraktionsebene	Niedrig – direkte Steuerung der Threads.	Höher – Fokus auf *was* getan werden soll, nicht *wie*.

Task-Klasse

Die `Task`-Klasse bildet einen Vorgang zur Lösung einer einzelnen Aufgabe ab, der keinen Wert zurück gibt und (in der Regel) asynchron ausgeführt wird. Ein `Task`-Objekt übernimmt eine Aufgabe, die asynchron auf einem Threadpool-Thread anstatt synchron auf dem Hauptanwendungsthread ausgeführt wird. Zum Überwachen des Bearbeitungsstatus stehen die Status-Eigenschaften des Threads und die Eigenschaften der Klasse `Task` zur Verfügung: `IsCanceled`, `IsCompleted`, und `IsFaulted`.

TaskClasses

```
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();
  ...
}
```

Instanziierung und Ausführung von Tasks:

Die Instanziirung erfolgt über einen Konstruktor, die Ausführung wird durch den Aufruf der Methode Start veranlasst:

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

> Bis hierher ist die API völlig identisch zu einem Thread (abgesehen von den Typen).

Der verkürzte Aufruf mittels der statischen `Run`-Methode realisiert das gleiche Verhalten:

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

An den Konstruktor und die Run-Methode können `Action`-Delegate übergeben werden, die den auszuführenden Code beinhalten.

> Delegaten können durch konkrete Methoden, anonyme Methoden oder Lambda-Ausdrücke realisiert werden.

Der Konstruktor wird nur in erweiterten Szenarien verwendet, wo es erforderlich ist, die Instanziirung und den Start zu trennen.

Überwachung

Über Property `IsCompleted` kann der laufende Task aus dem Main-Thread überwacht werden. Um für die Durchführung einer einzelnen Aufgabe zu warten, rufen Sie die `Task.Wait` Methode auf. Ein Aufruf der Wait-Methode blockiert den aufrufenden Thread, bis die Instanz der Klasse die Ausführung abgeschlossen hat.

TaskDefinition1

```
1  // Motiviert aus
2  // https://docs.microsoft.com/de-de/dotnet/api/system.threading.tasks
     ?view=netframework-4.8
3  using System;
4  using System.Threading.Tasks;
5  using System.Threading;
6
7  public class Example
8  {
9     public static void doSomething(){
10      Console.WriteLine("Say hello!");
11    }
12
13    public static void Main()
14    {
15       Action<object> action = (object obj) =>
16                               {
17                                  Console.WriteLine("Task={0}, obj={1}, Thre
                                      ={2}",
18                                  Task.CurrentId, obj,
19                                  Thread.CurrentThread.ManagedThreadId);
20                               };
21
22       Task t1 = new Task(action, "alpha");
23       t1.Start();
24       Console.WriteLine("t1 has been launched. (Main Thread={0})",
25                                  Thread.CurrentThread.ManagedThreadId)
26
27       // Nur der Vollständigkeit halber ...
28       Task t2 = new Task(doSomething);
29       t2.Start();
30       Console.WriteLine("t2 has been launched. (Main Thread={0})",
31                                  Thread.CurrentThread.ManagedThreadId)

32
33
34       Task t3 = Task.Run( () => {
35                               // Just loop.
36                               int ctr = 0;
37                               for (ctr = 0; ctr <= 1000000; ctr++)
38                               {}
39                               Console.WriteLine("Finished {0} loop iteratio
40                                                 ctr);
41                            } );
42       t3.Wait();
43    }
44 }
```

`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 oder ein Timeoutintervall abgelaufen ist, je nach dem welcher Fall zuerst eintritt.

WaitForNTimeSlots

```
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));
    //Task taskX = Task.Run(() => { throw new IndexOutOfRangeExceptio
      );
    //Task taskY = Task.Run(() => { throw new FormatException(); } );
    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.");
      //taskX.Wait();
      //taskY.Wait();
     }
     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.

Im folgenden Beispiel werden zehn Aufgaben erstellt, die warten, bis alle zehn abgeschlossen werden, dann wird ihr Status anzeigt.

WaitForAll

```
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);
    }
}
```

Generische Task-Klasse

Die generische Klasse `Task<T>` bildet ebenfalls einen Vorgang zur Lösung einer einzelnen Aufgabe ab, gibt aber im Unterschied zu der nicht generischen `Task`-Klasse einen Wert zurück. Die Konstruktoren und die Run-Methode der Klasse bekommen einen `Func`-Delegat bzw. einen als Lambda-Ausdruck formulierten Code übergeben, der einen Rückgabewert liefert.

TaskClasses

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

Der Kanon der Möglichkeiten wird durch die Klasse `Task<TResult>` deutlich erweitert. Anstatt die Ergebnisse wie bei Threads in eine "außen stehende" Variable (z.B. Datenfeld einer Klasse) zu speichern, wird das Ergebnis im `Task`-Objekt selbst gespeichert und kann dann über die Eigenschaft `Result` abgerufen werden.

TaskWithReturn

```
Task<int> task = Task.Run(() => {int i;
                                 //... Anweisungsblock ...;
                                 return i;});
 Console.WriteLine("Finished with result {0}", task.Result);
```

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 und wann?

Asynchrone Methoden

Das Kernstück der asynchronen Programmierung mit TAP (task based asynchronous pattern) sind die Schlüsselwörter `async` und `await`. "async" wird verwendet, um eine Methode zu markieren, die asynchronen Code enthält, "await" wird verwendet, um auf das Ergebnis einer asynchronen Operation zu warten, ohne den Aufrufer zu blockieren.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {   Console.WriteLine("Beispiel mit Download");
        int n=await DownloadFileAsync();
        Console.WriteLine(n);
        Console.WriteLine("Download abgeschlossen!");
    }

    public static async Task<int> DownloadFileAsync()
    {
        using (var httpClient = new HttpClient())
        {
            Console.WriteLine("Starte den Download...");
            var url = "https://github.com/TUBAF-IfI-LiaScript
              /VL_Softwareentwicklung/blob/master/24_Tasks.md";
            var response = await httpClient.GetAsync(url);
            var content = await response.Content.ReadAsStringAsync();
```

```
            //Console.WriteLine("Datei heruntergeladen: " + content);
            return content.Length;
        }
    }
}
```

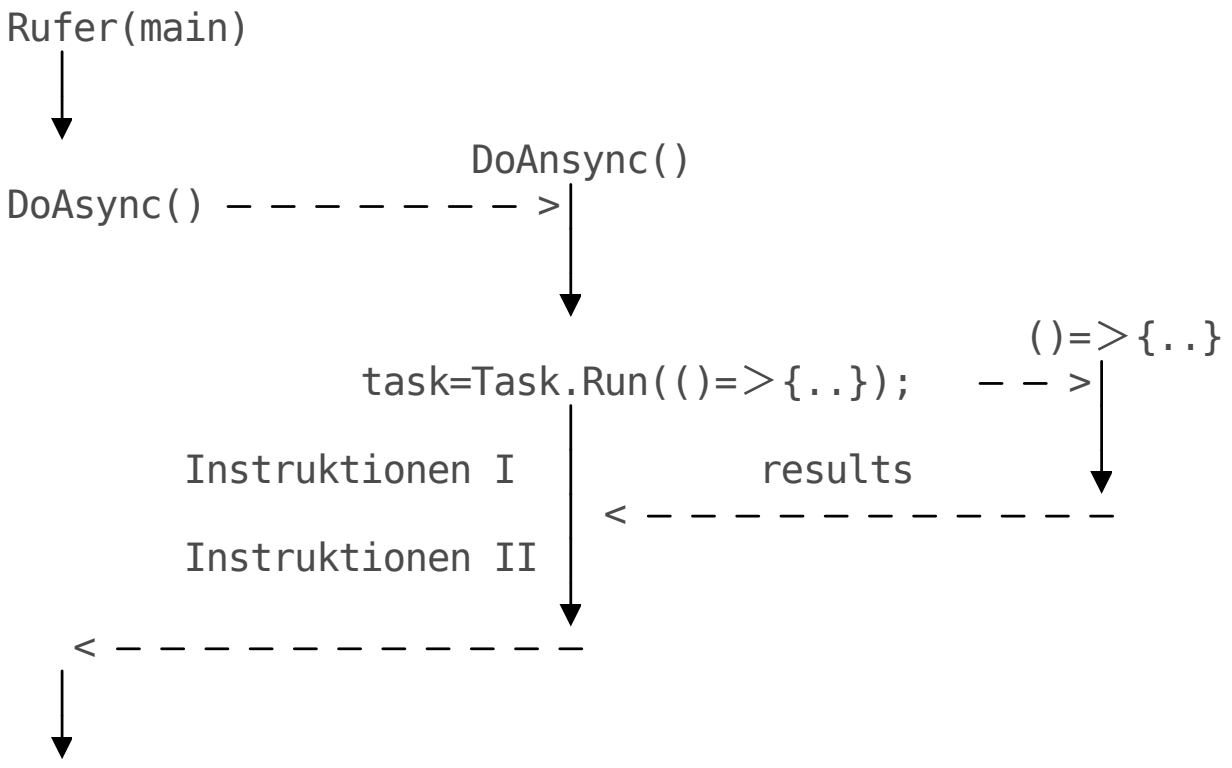
Die Initiierung und der Abschluss eines asynchronen Vorgangs wird in TAP in einer Methode realisiert, die das `async`-Präfix hat und dadurch eine `await`-Anweisung enthalten darf, wenn sie Awaitable-Typen zurückgibt, wie z. B. Task oder Task<TResult>.

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

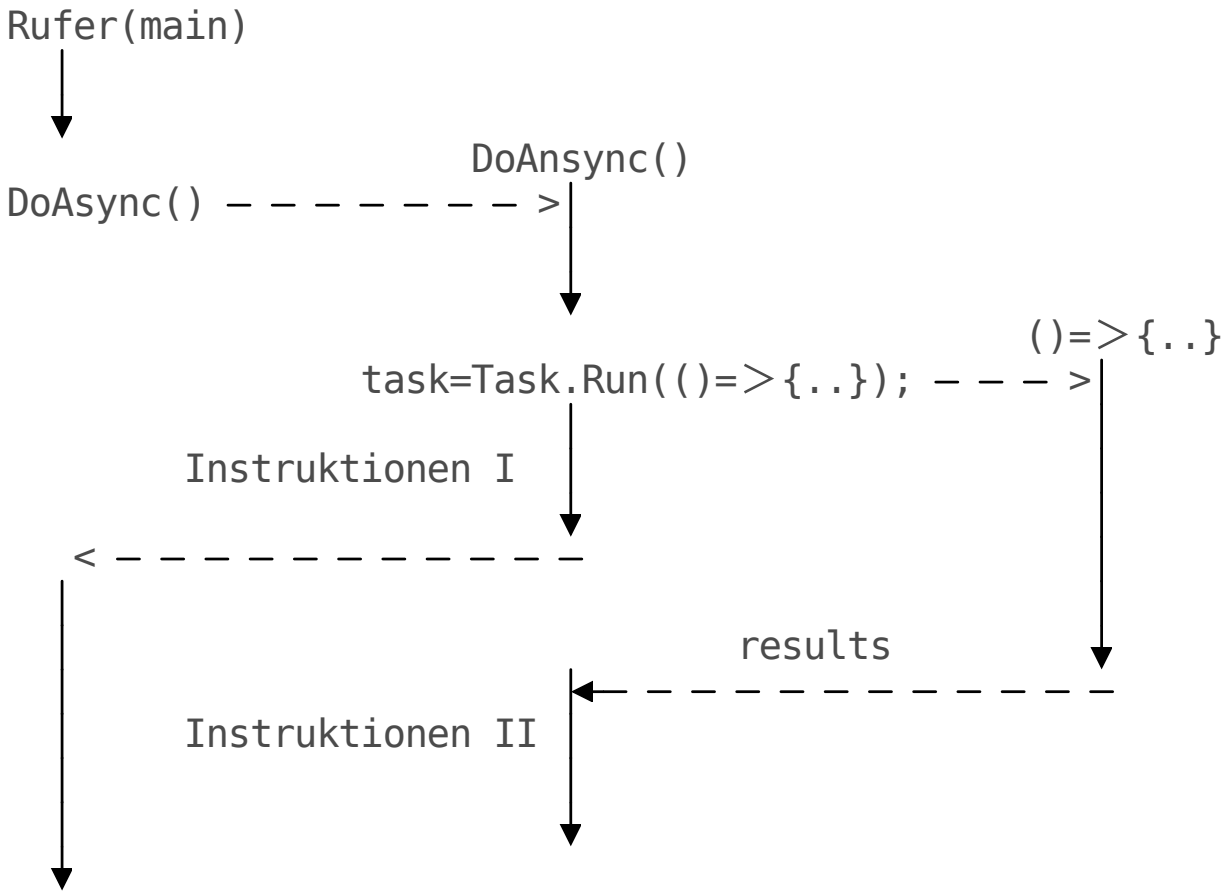
```
async void DoAsync(){
  Task<int> task = Task.Run(() => {int i;
                                   // Berechnungen
                                   return i;});
  // Instruktionen I
  // Methoden, die unabhängig von task ausgeführt werden
  int result = await task;
  // Instruktionen II
  // Hier 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. Die Methode pausiert an der Stelle des await-Ausdrucks und wartet darauf, dass der Task abgeschlossen wird. Während dieser Wartezeit wird der Thread, auf dem DoAsync() ausgeführt wird, nicht blockiert, sondern steht für andere Aufgaben zur Verfügung.



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

Beispiel	**Bemerkung**
AsyncExampleI.cs	Generelle Einbettung des asynchronen Tasks
AsyncExampleII.cs	Was passiert eigentlich, wenn `main` zum Ende kommt mit den noch ausbleibenden Ergebnissen von Tasks?