Async C#

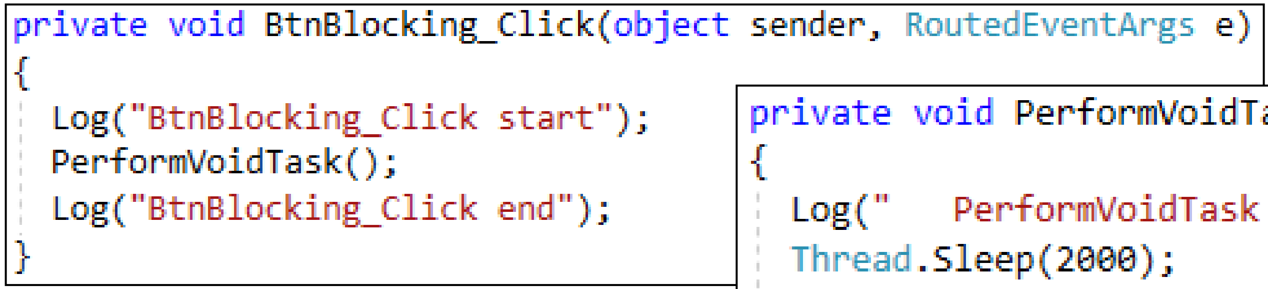
Thread
Task
async/await
TPL – Task Parallel Library

Asynchrone Programmierung

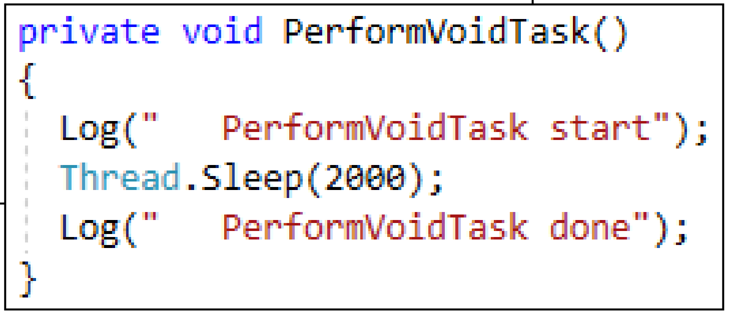
- Methoden gleichzeitig ausführen
- vor allem bei Desktop Apps
  - Synchrone Methode blockiert UI Thread
  - App reagiert auf keine User-Eingaben
- Mehrere Konzepte
  - Thread
  - BackgroundWorker (ähnlich AsyncTask bei Android)
  - Task
  - async/await

Synchron

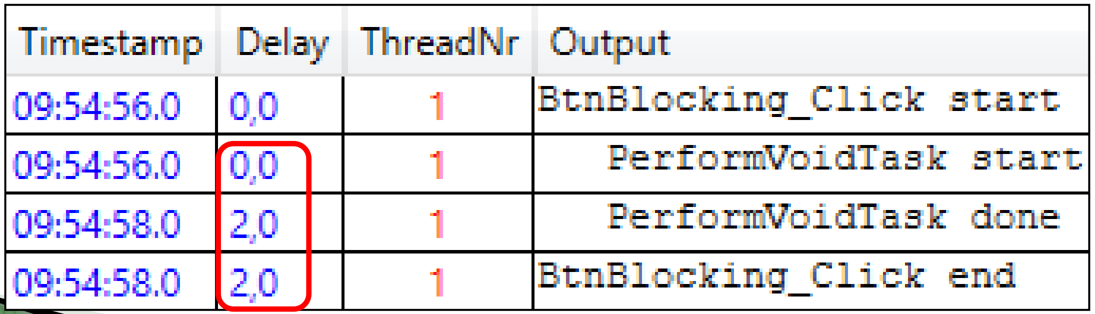
- Blockiert den Aufrufer
  - Simuliert durch Thread.Sleep
  - ➔App ist 2 Sekunden nicht bedienbar



```
private void BtnBlocking_Click(object sender, RoutedEventArgs e)
{
    Log("BtnBlocking_Click start");
    PerformVoidTask();
    Log("BtnBlocking_Click end");
}
```



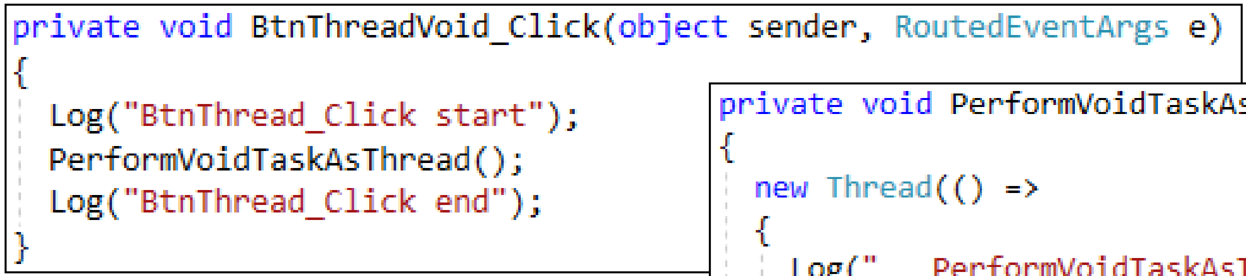
```
private void PerformVoidTask()
{
    Log("    PerformVoidTask start");
    Thread.Sleep(2000);
    Log("    PerformVoidTask done");
}
```



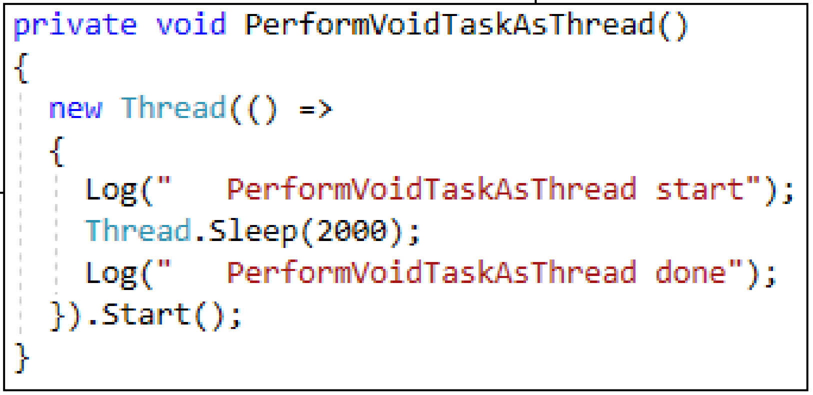
Timestamp	Delay	ThreadNr	Output
09:54:56.0	0,0	1	BtnBlocking_Click start
09:54:56.0	0,0	1	PerformVoidTask start
09:54:58.0	2,0	1	PerformVoidTask done
09:54:58.0	2,0	1	BtnBlocking_Click end

Thread

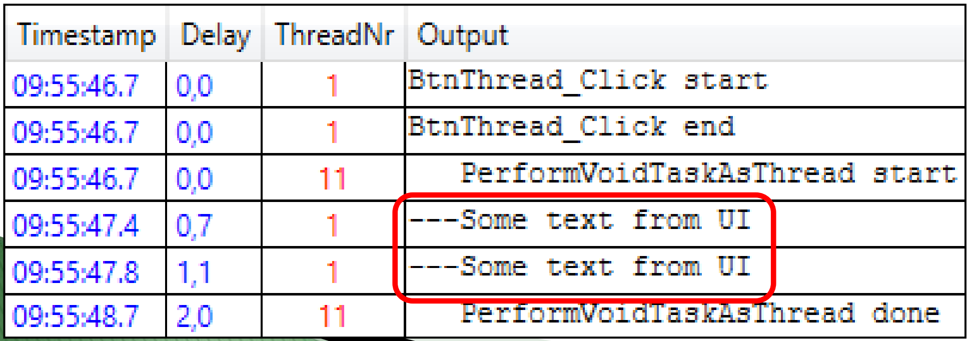
- Blockiert den Aufrufer nicht
  - Starten ähnlich wie in Java: new Thread().Start()
  - ➔App bleibt bedienbar



```
private void BtnThreadVoid_Click(object sender, RoutedEventArgs e)
{
    Log("BtnThread_Click start");
    PerformVoidTaskAsThread();
    Log("BtnThread_Click end");
}
```



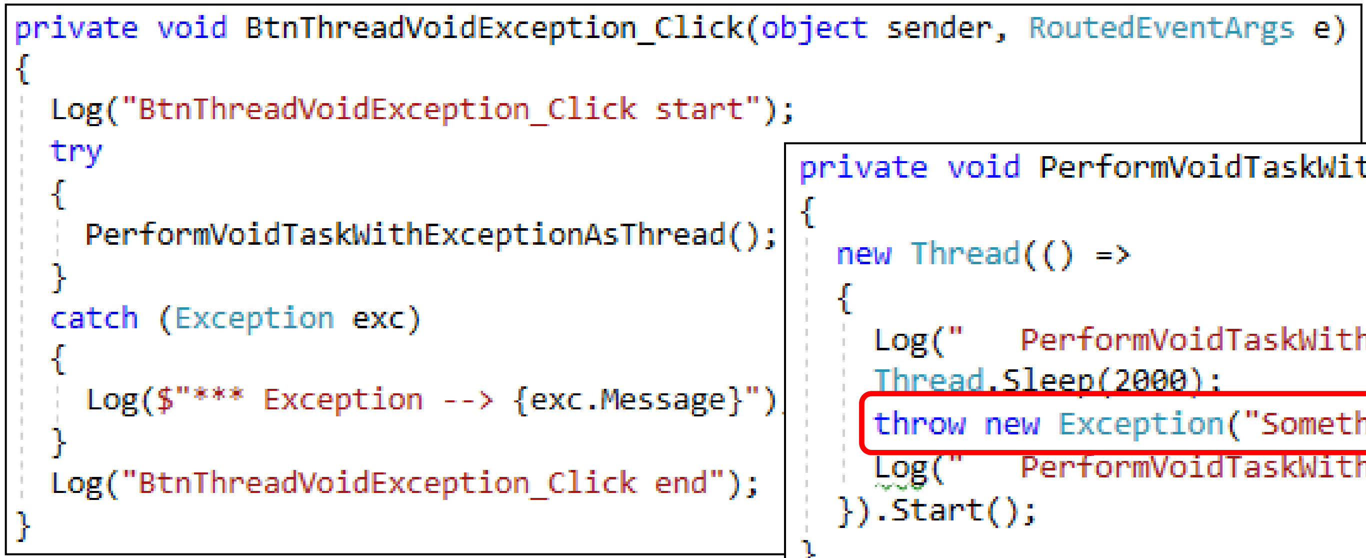
```
private void PerformVoidTaskAsThread()
{
    new Thread(() =>
    {
        Log("    PerformVoidTaskAsThread start");
        Thread.Sleep(2000);
        Log("    PerformVoidTaskAsThread done");
    }).Start();
}
```



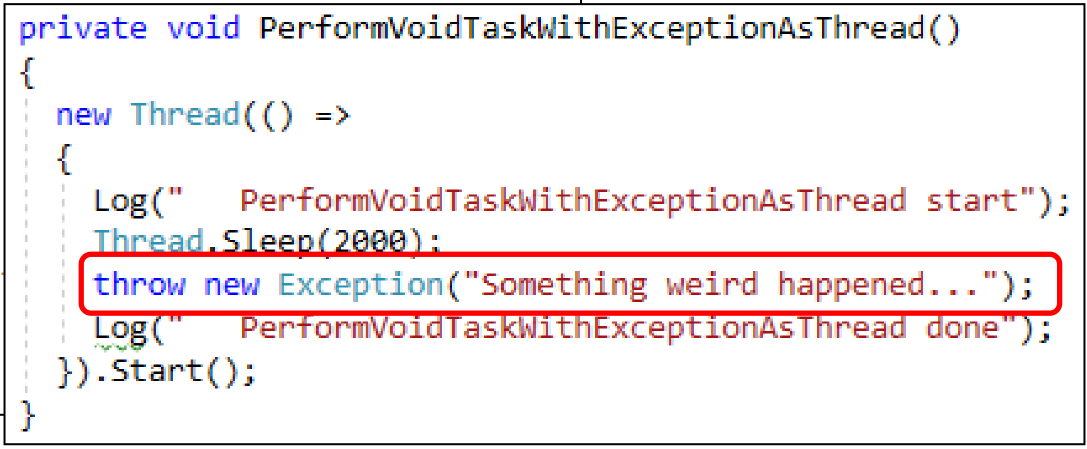
Timestamp	Delay	ThreadNr	Output
09:55:46.7	0,0	1	BtnThread_Click start
09:55:46.7	0,0	1	BtnThread_Click end
09:55:46.7	0,0	11	PerformVoidTaskAsThread start
09:55:47.4	0,7	1	---Some text from UI
09:55:47.8	1,1	1	---Some text from UI
09:55:48.7	2,0	11	PerformVoidTaskAsThread done

Thread Problem 1: Exception

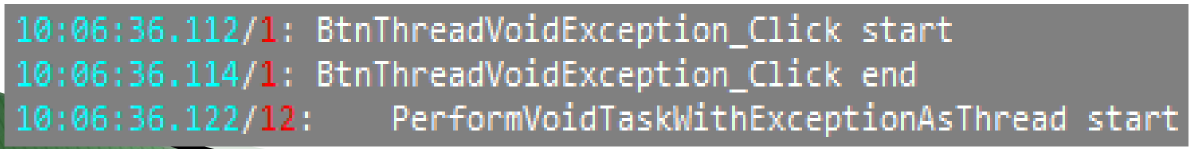
- Aufrufer exekutiert weiter
- Ist über etwaigen catch-Block hinweg
- ➔ kann keine Exception fangen



```
private void BtnThreadVoidException_Click(object sender, RoutedEventArgs e)
{
  Log("BtnThreadVoidException_Click start");
  try
  {
    PerformVoidTaskWithExceptionAsThread();
  }
  catch (Exception exc)
  {
    Log($"*** Exception --> {exc.Message}")
  }
  Log("BtnThreadVoidException_Click end");
}
```



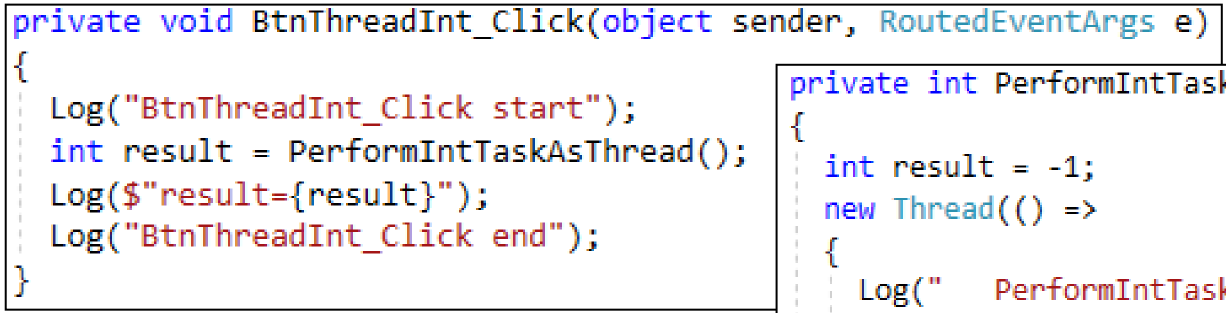
```
private void PerformVoidTaskWithExceptionAsThread()
{
  new Thread(() =>
  {
    Log("    PerformVoidTaskWithExceptionAsThread start");
    Thread.Sleep(2000);
    throw new Exception("Something weird happened...");
    Log("    PerformVoidTaskWithExceptionAsThread done");
  }).Start();
}
```



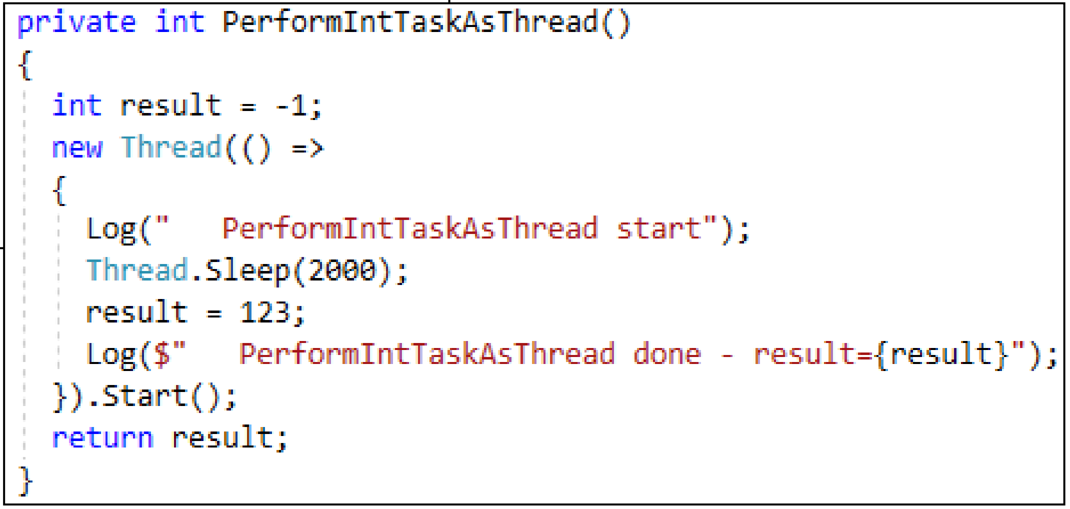
```
10:06:36.112/1: BtnThreadVoidException_Click start
10:06:36.114/1: BtnThreadVoidException_Click end
10:06:36.122/12:    PerformVoidTaskWithExceptionAsThread start
```

Thread Problem 2: return value

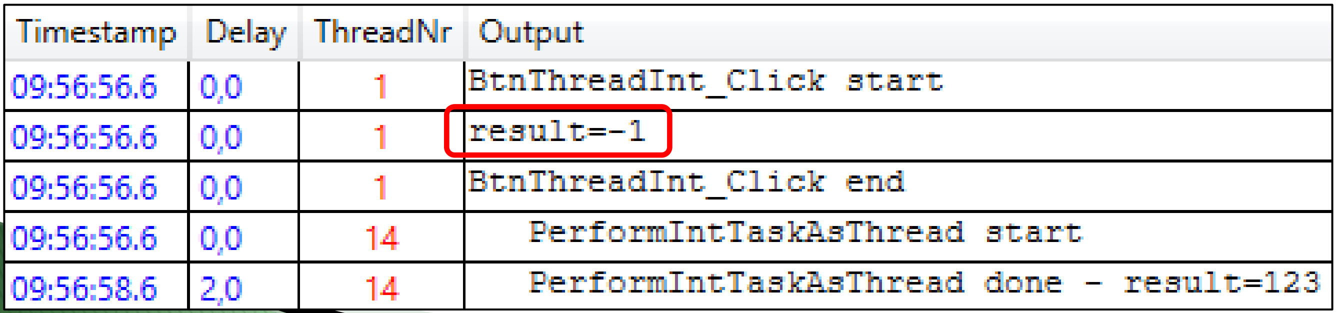
- Rückgabewert ist nicht verfügbar
  - zumindest nicht ohne deutlichen Zusatzaufwand



```csharp
private void BtnThreadInt_Click(object sender, RoutedEventArgs e)
{
  Log("BtnThreadInt_Click start");
  int result = PerformIntTaskAsThread();
  Log($"result={result}");
  Log("BtnThreadInt_Click end");
}
```



```csharp
private int PerformIntTaskAsThread()
{
  int result = -1;
  new Thread(() =>
  {
    Log("   PerformIntTaskAsThread start");
    Thread.Sleep(2000);
    result = 123;
    Log($"   PerformIntTaskAsThread done - result={result}");
  }).Start();
  return result;
}
```



Timestamp	Delay	ThreadNr	Output
09:56:56.6	0,0	1	BtnThreadInt_Click start
09:56:56.6	0,0	1	result=-1
09:56:56.6	0,0	1	BtnThreadInt_Click end
09:56:56.6	0,0	14	PerformIntTaskAsThread start
09:56:58.6	2,0	14	PerformIntTaskAsThread done - result=123

Task

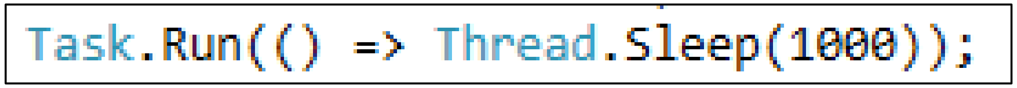
- Löst die geschilderten Probleme
- Konvention: Methodenname endet mit Async
- Startet asynchronen Code mit `Task.Run()`
  - verhält sich ähnlich wie `new Thread().Start()`
- Rückgabewert ist ein Task-Objekt
- Auf Ergebnis warten
  - `Wait()`
  - `Result`
  - beide blockieren aber ebenfalls!
- Exceptions werden zum Aufrufer transportiert
  - und zwar als `AggregateException`
  - `InnerException` enthält tatsächliche Exception

Klasse Task

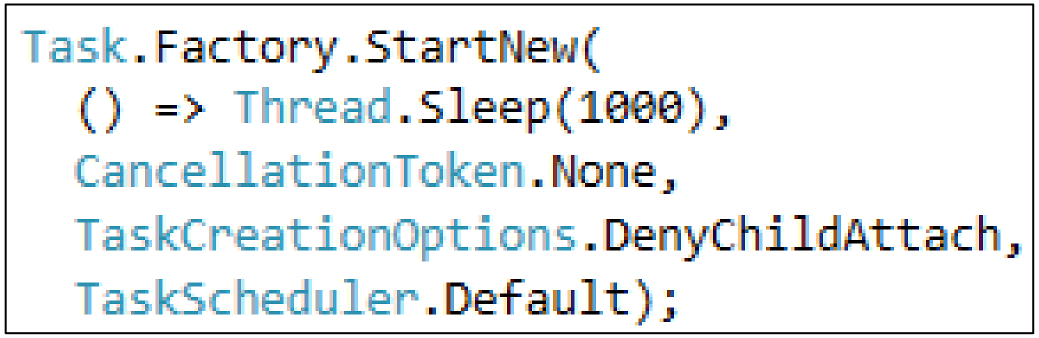
```
namespace System.Threading.Tasks
{
  public class Task : IAsyncResult, IDisposable
  {
    public Task(Action action);
    public static Task Run(Action action);
    public void Wait();
    public bool IsCompleted { get; }
    public TaskStatus Status { get; }
    …
  }
}
```

Task starten

- Am einfachsten mit **`Task.Run(action)`**
- Oder **`Task.Factory.StartNew(action)`**
  - hat 16 Überladungen
- Folgendes ist gleichwertig



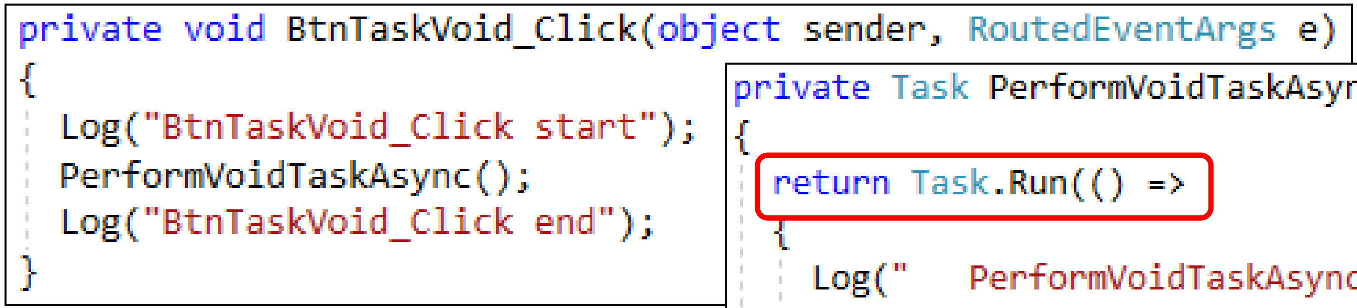
```
Task.Run(() => Thread.Sleep(1000));
```



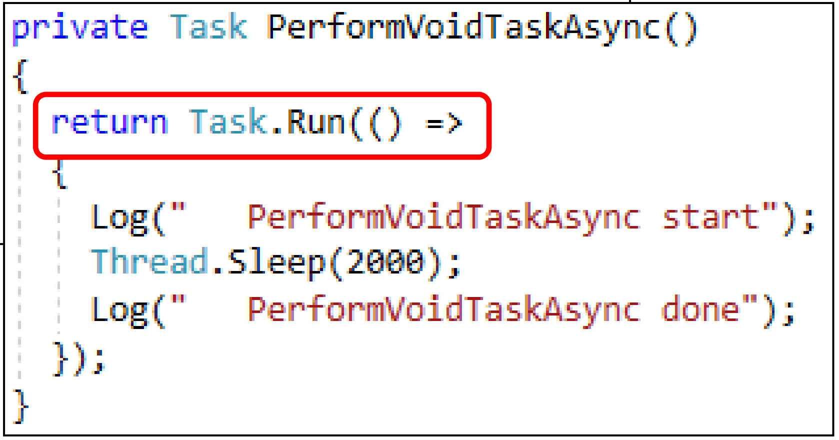
```
Task.Factory.StartNew(
    () => Thread.Sleep(1000),
    CancellationToken.None,
    TaskCreationOptions.DenyChildAttach,
    TaskScheduler.Default);
```

Task starten

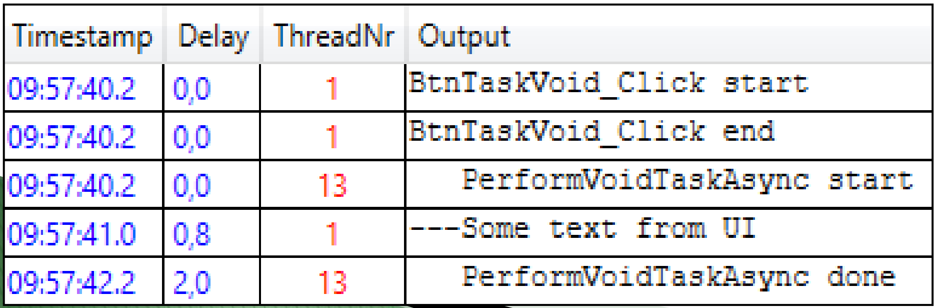
- Blockiert nicht ➔ App bleibt bedienbar
- Läuft in eigenem Thread
- Returntyp: Task-Objekt



```
private void BtnTaskVoid_Click(object sender, RoutedEventArgs e)
{
    Log("BtnTaskVoid_Click start");
    PerformVoidTaskAsync();
    Log("BtnTaskVoid_Click end");
}
```



```
private Task PerformVoidTaskAsync()
{
    return Task.Run(() =>
    {
        Log("    PerformVoidTaskAsync start");
        Thread.Sleep(2000);
        Log("    PerformVoidTaskAsync done");
    });
}
```

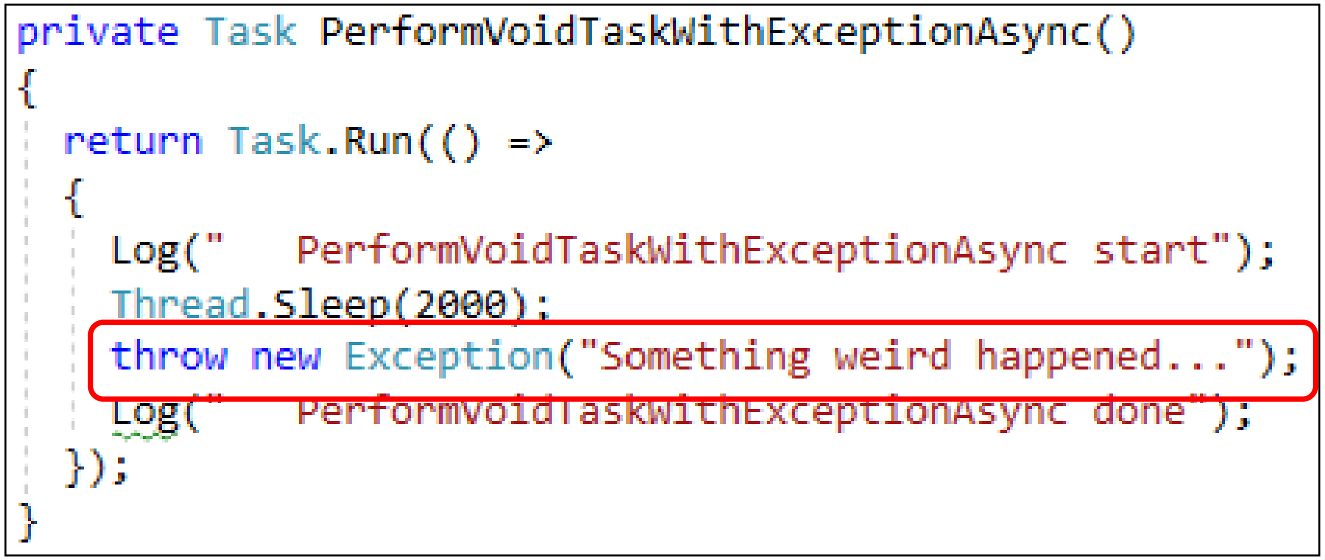


Timestamp	Delay	ThreadNr	Output
09:57:40.2	0,0	1	BtnTaskVoid_Click start
09:57:40.2	0,0	1	BtnTaskVoid_Click end
09:57:40.2	0,0	13	PerformVoidTaskAsync start
09:57:41.0	0,8	1	---Some text from UI
09:57:42.2	2,0	13	PerformVoidTaskAsync done

Achtung: Ist noch nicht die richtige Lösung!

Task mit Exceptions

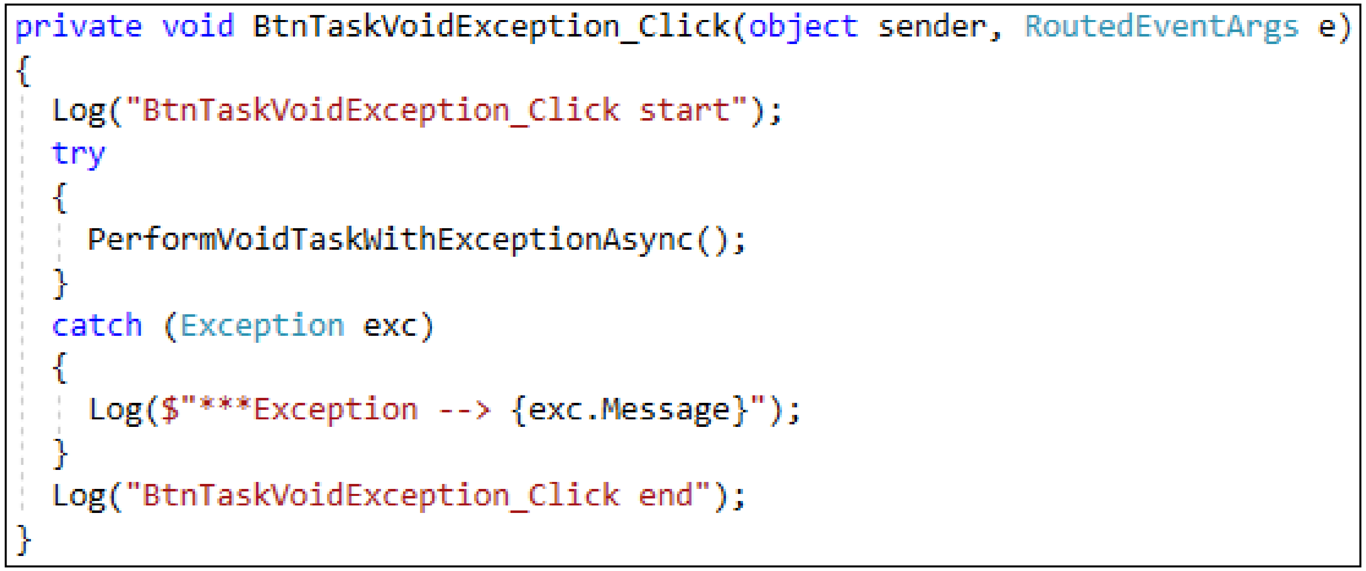
Exceptions werfen wie gewohnt



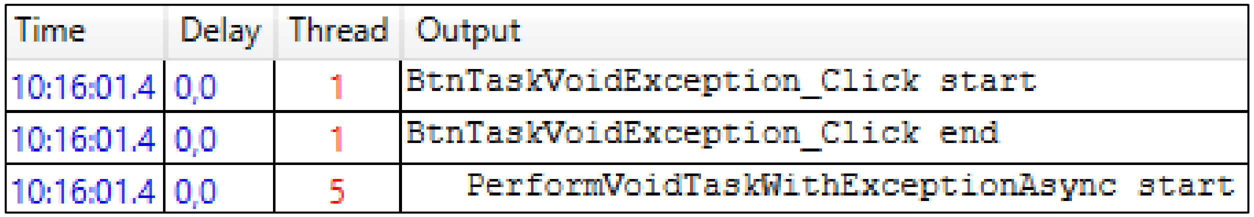
```
private Task PerformVoidTaskWithExceptionAsync()
{
    return Task.Run(() =>
    {
        Log("    PerformVoidTaskWithExceptionAsync start");
        Thread.Sleep(2000);
        throw new Exception("Something weird happened...");
        Log("    PerformVoidTaskWithExceptionAsync done");
    });
}
```

Task catch Exceptions v1

- Folgender Code funktioniert nicht



```
private void BtnTaskVoidException_Click(object sender, RoutedEventArgs e)
{
    Log("BtnTaskVoidException_Click start");
    try
    {
        PerformVoidTaskWithExceptionAsync();
    }
    catch (Exception exc)
    {
        Log($"***Exception --> {exc.Message}");
    }
    Log("BtnTaskVoidException_Click end");
}
```

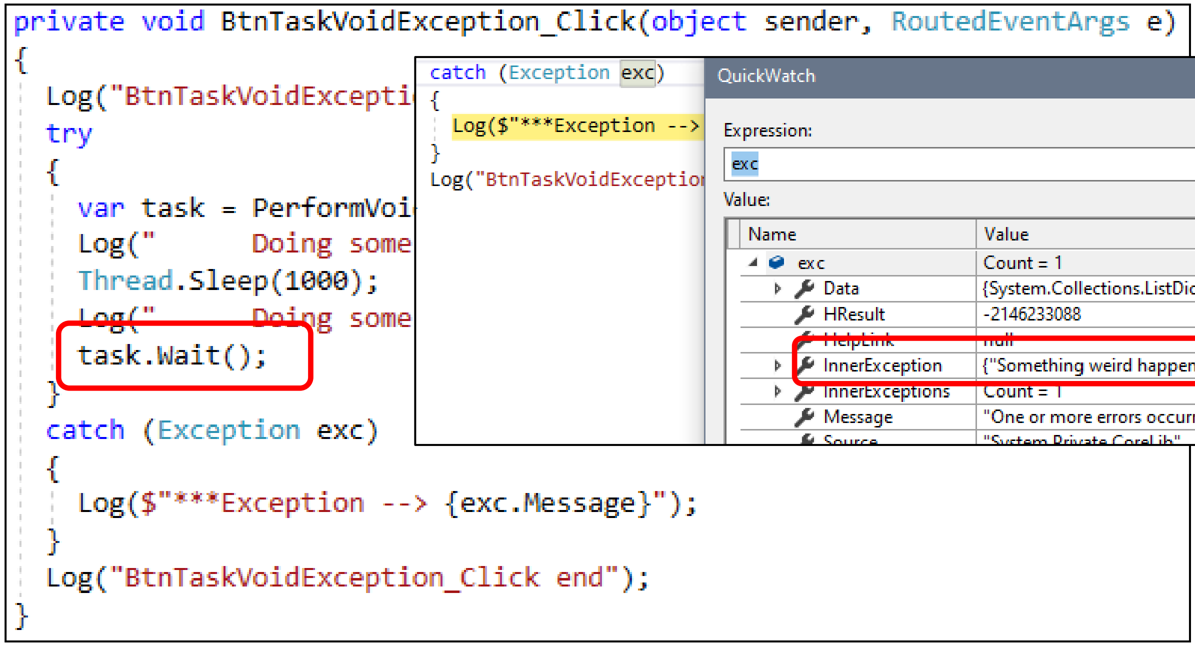


Time	Delay	Thread	Output
10:16:01.4	0,0	1	BtnTaskVoidException_Click start
10:16:01.4	0,0	1	BtnTaskVoidException_Click end
10:16:01.4	0,0	5	PerformVoidTaskWithExceptionAsync start

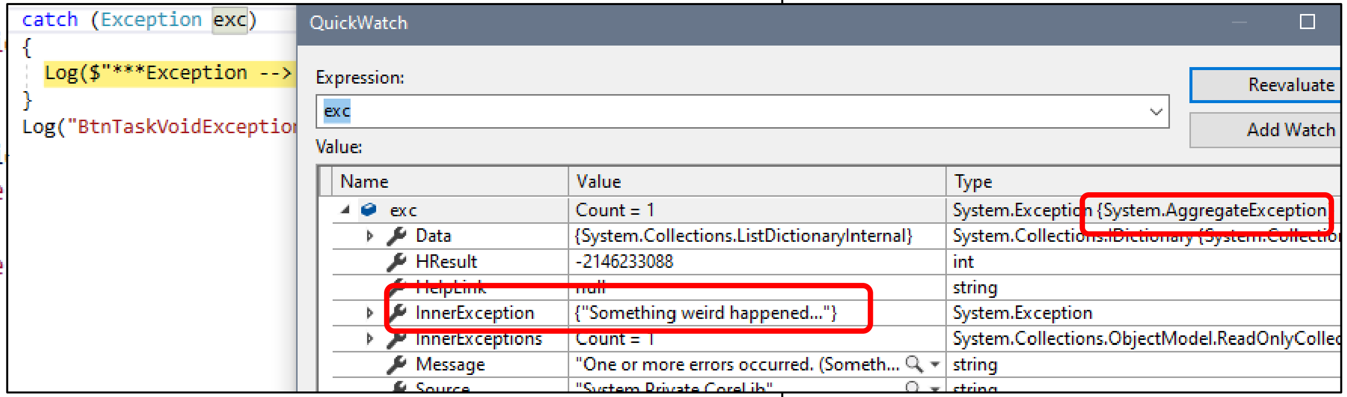
- Problem: Aufrufer wartet nicht!

Task catch Exceptions v2

- Warten mit **task.Wait()**



```
private void BtnTaskVoidException_Click(object sender, RoutedEventArgs e)
{
  Log("BtnTaskVoidExcepti
  try
  {
    var task = PerformVoi
    Log("       Doing some
    Thread.Sleep(1000);
    Log("       Doing some
    task.Wait();
  }
  catch (Exception exc)
  {
    Log($"***Exception --> {exc.Message}");
  }
  Log("BtnTaskVoidException_Click end");
}
```



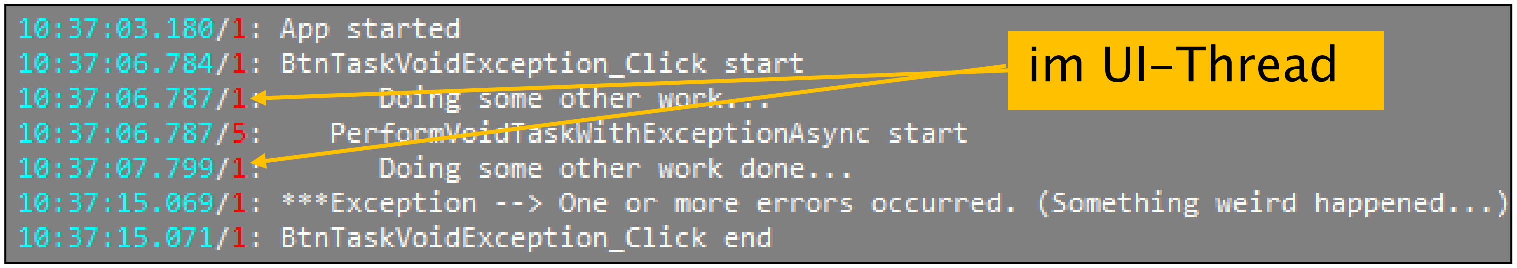
```
catch (Exception exc)
{
  Log($"***Exception -->
}
Log("BtnTaskVoidExceptio
```

QuickWatch

Expression: exc

Value:

Name	Value	Type
exc	Count = 1	System.Exception {System.AggregateException}
Data	{System.Collections.ListDictionaryInternal}	System.Collections.IDictionary {System.Collectio
HResult	-2146233088	int
HelpLink	null	string
InnerException	{"Something weird happened..."}	System.Exception
InnerExceptions	Count = 1	System.Collections.ObjectModel.ReadOnlyCollec
Message	"One or more errors occurred. (Someth...	string
Source	"System.Private.CoreLib"	string

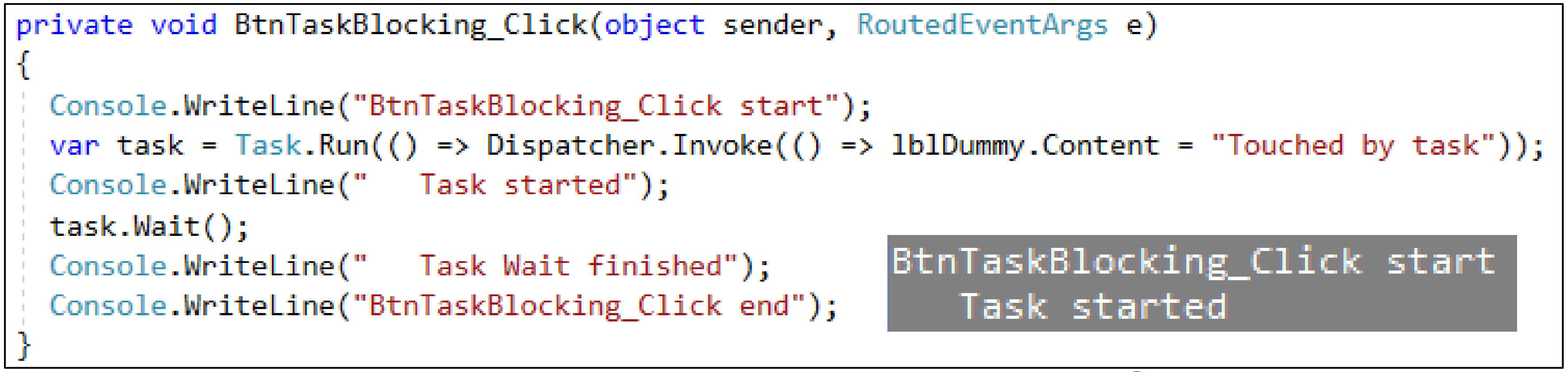


```
10:37:03.180/1: App started
10:37:06.784/1: BtnTaskVoidException_Click start
10:37:06.787/1:       Doing some other work...
10:37:06.787/5:    PerformVoidTaskWithExceptionAsync start
10:37:07.799/1:       Doing some other work done...
10:37:15.069/1: ***Exception --> One or more errors occurred. (Something weird happened...)
10:37:15.071/1: BtnTaskVoidException_Click end
```

Achtung: Mit Dispatcher.Invoke() ➔ deadlock!

Task deadlock

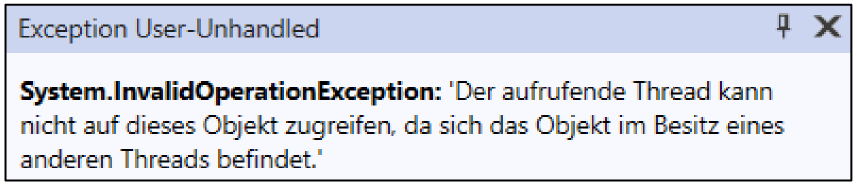
- Folgender Code führt zu Deadlock



```
private void BtnTaskBlocking_Click(object sender, RoutedEventArgs e)
{
  Console.WriteLine("BtnTaskBlocking_Click start");
  var task = Task.Run(() => Dispatcher.Invoke(() => lblDummy.Content = "Touched by task"));
  Console.WriteLine("   Task started");
  task.Wait();
  Console.WriteLine("   Task Wait finished");
  Console.WriteLine("BtnTaskBlocking_Click end");
}
```

```
BtnTaskBlocking_Click start
   Task started
```

- Grund: Zwei Codeteile warten auf UI-Thread
  - Dispatcher.Invoke
  - task.Wait
- Ohne Dispatcher:

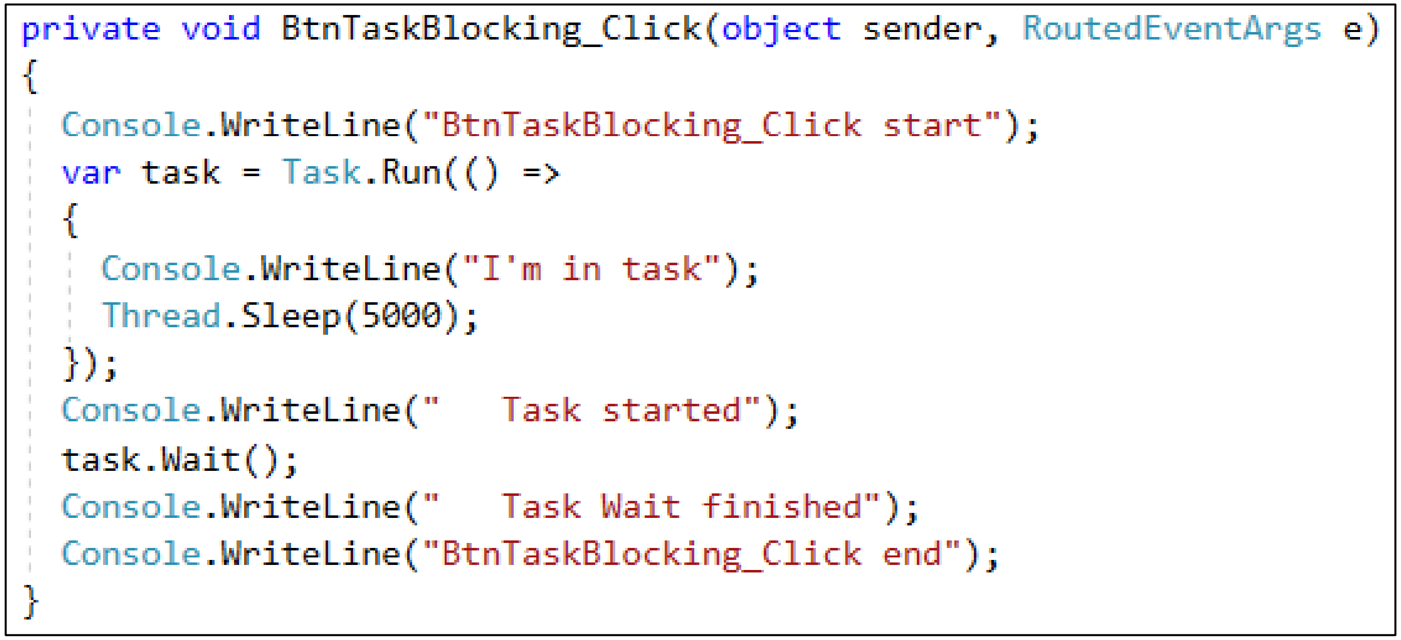


Exception User-Unhandled

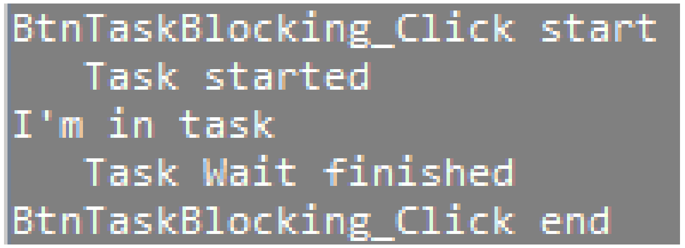
System.InvalidOperationException: 'Der aufrufende Thread kann nicht auf dieses Objekt zugreifen, da sich das Objekt im Besitz eines anderen Threads befindet.'

Task.Wait blockiert!

- Folgender Code blockiert für 5 Sekunden



```
private void BtnTaskBlocking_Click(object sender, RoutedEventArgs e)
{
  Console.WriteLine("BtnTaskBlocking_Click start");
  var task = Task.Run(() =>
  {
    Console.WriteLine("I'm in task");
    Thread.Sleep(5000);
  });
  Console.WriteLine("   Task started");
  task.Wait();
  Console.WriteLine("   Task Wait finished");
  Console.WriteLine("BtnTaskBlocking_Click end");
}
```



```
BtnTaskBlocking_Click start
   Task started
I'm in task
   Task Wait finished
BtnTaskBlocking_Click end
```

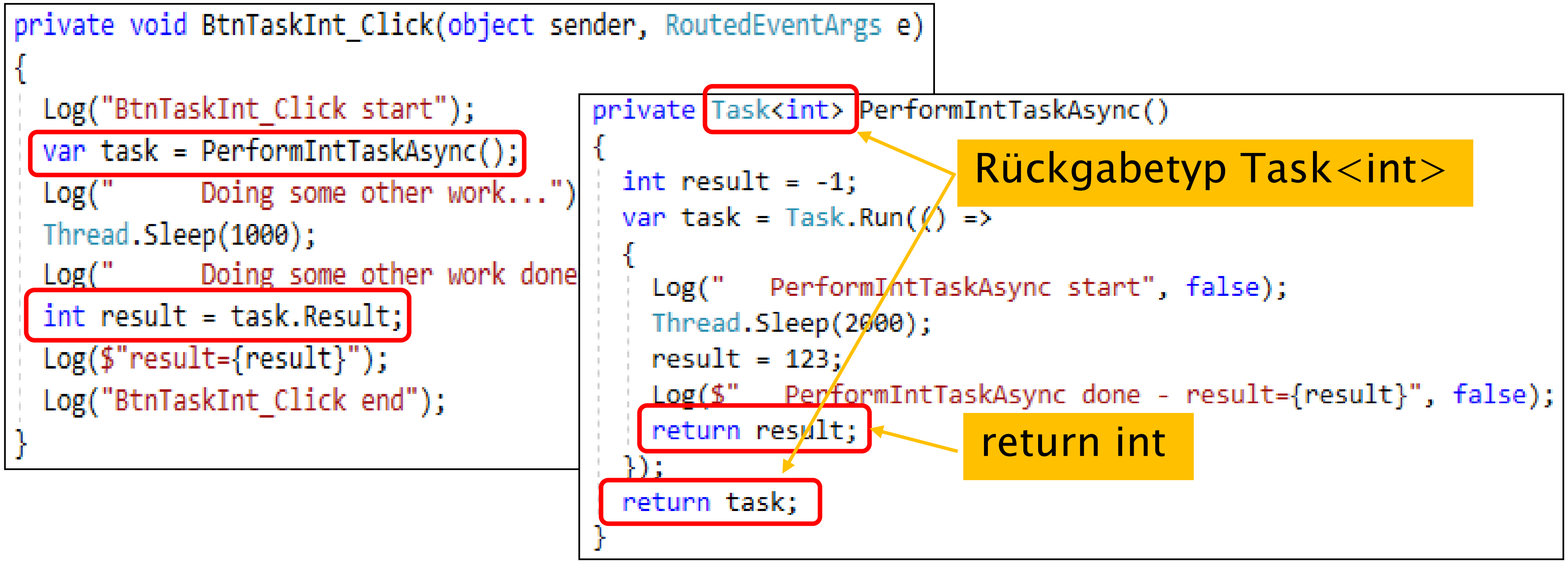
Task mit Rückgabewert

- Rückgabetyp ist **`Task<T>`**
- **`Task.Run`** mit Parameter **`Func<T>`**
- Innerhalb von Task.Run wird daher Typ **`T`** zurückgegeben
- Aufrufer erhält Wert über **`task.Result`**
  - ruft intern task.Wait() auf
- Auch hier wieder: task.Result blockiert
- ➔Deadlock, wenn man mit Dispatcher.Invoke auf UI-Thread zugreift

Klasse Task<T>

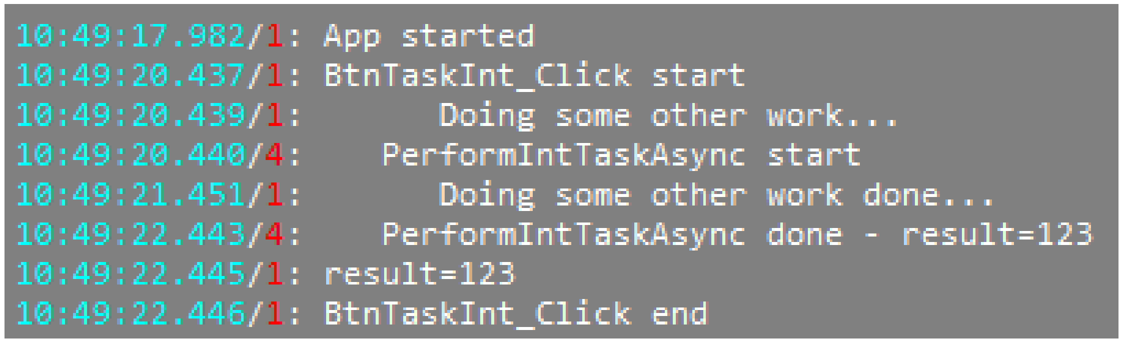
```
namespace System.Threading.Tasks
{
  public class Task<TResult> : Task
  {
    public Task(Func<TResult> function);
    public TResult Result { get; }
    …
  }
}
```

Task mit int



```
private void BtnTaskInt_Click(object sender, RoutedEventArgs e)
{
    Log("BtnTaskInt_Click start");
    var task = PerformIntTaskAsync();
    Log("        Doing some other work...")
    Thread.Sleep(1000);
    Log("        Doing some other work done
    int result = task.Result;
    Log($"result={result}");
    Log("BtnTaskInt_Click end");
}
```

```
private Task<int> PerformIntTaskAsync()
{
    int result = -1;
    var task = Task.Run(() =>
    {
        Log("    PerformIntTaskAsync start", false);
        Thread.Sleep(2000);
        result = 123;
        Log($"    PerformIntTaskAsync done - result={result}", false);
        return result;
    });
    return task;
}
```

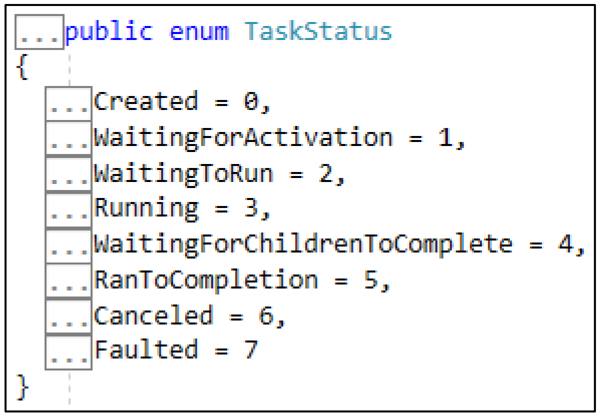


```
10:49:17.982/1: App started
10:49:20.437/1: BtnTaskInt_Click start
10:49:20.439/1:         Doing some other work...
10:49:20.440/4:     PerformIntTaskAsync start
10:49:21.451/1:         Doing some other work done...
10:49:22.443/4:     PerformIntTaskAsync done - result=123
10:49:22.445/1: result=123
10:49:22.446/1: BtnTaskInt_Click end
```

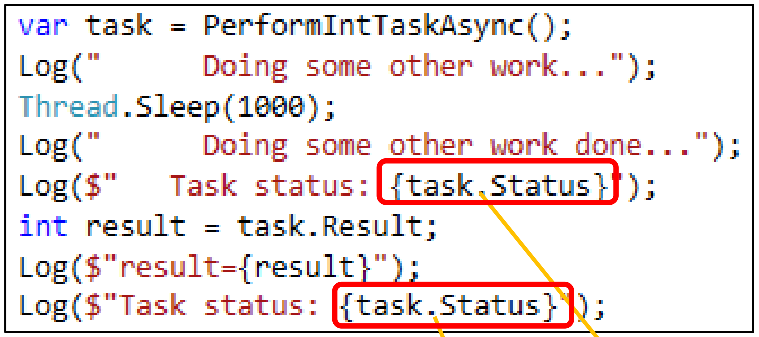
- Auch hier wieder: task.Result blockiert!

Task status

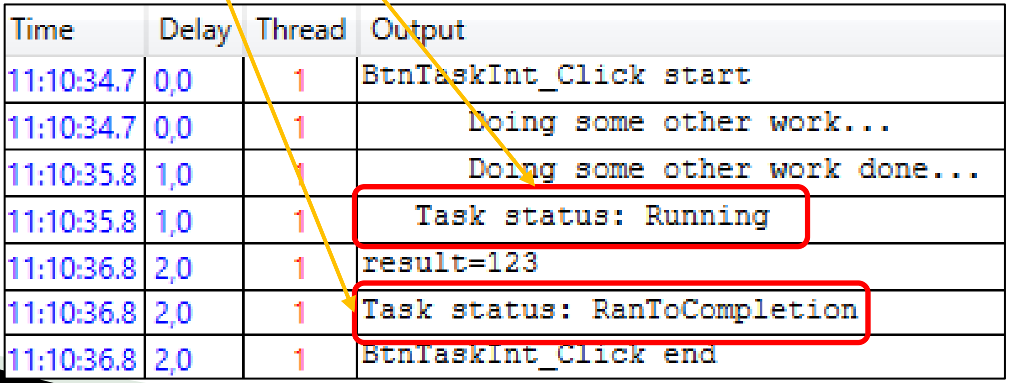
- Klasse **TaskStatus**



```
...public enum TaskStatus
{
    ...Created = 0,
    ...WaitingForActivation = 1,
    ...WaitingToRun = 2,
    ...Running = 3,
    ...WaitingForChildrenToComplete = 4,
    ...RanToCompletion = 5,
    ...Canceled = 6,
    ...Faulted = 7
}
```



```
var task = PerformIntTaskAsync();
Log("      Doing some other work...");
Thread.Sleep(1000);
Log("      Doing some other work done...");
Log($"   Task status: {task.Status}");
int result = task.Result;
Log($"result={result}");
Log($"Task status: {task.Status}");
```

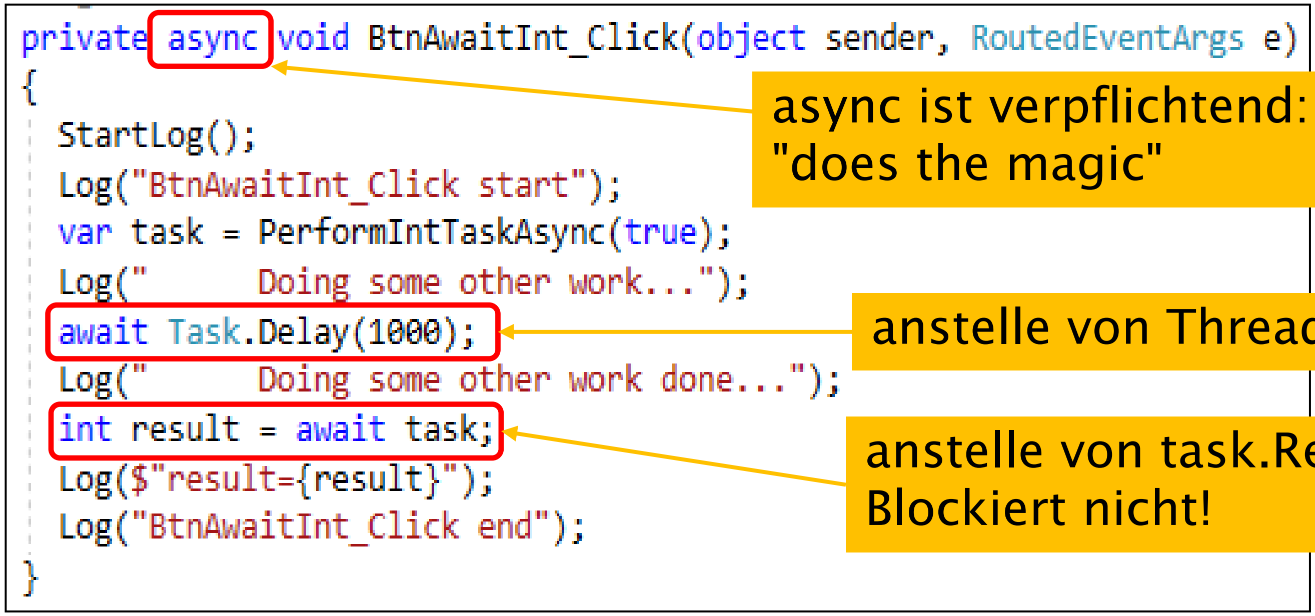


Time	Delay	Thread	Output
11:10:34.7	0,0	1	BtnTaskInt_Click start
11:10:34.7	0,0	1	Doing some other work...
11:10:35.8	1,0	1	Doing some other work done...
11:10:35.8	1,0	1	Task status: Running
11:10:36.8	2,0	1	result=123
11:10:36.8	2,0	1	Task status: RanToCompletion
11:10:36.8	2,0	1	BtnTaskInt_Click end

async/await

- Vereinfachung der Syntax
- Anstelle von task.Wait() bzw. task.Result schreibt man **`await`** task
- Methoden mit await müssen **`async`** deklariert werden
- Compiler erzeugt aber völlig anderen Code (mit State Machine)
- await erfordert awaitable: **`Task`** bzw. **`Task<T>`**
- Unterbricht, bis awaitable completed ist
- fährt im Ursprungskontext fort: "Continuation"
- AggregateException wird automatisch in Original-Exception umgewandelt

async/await Beispiel

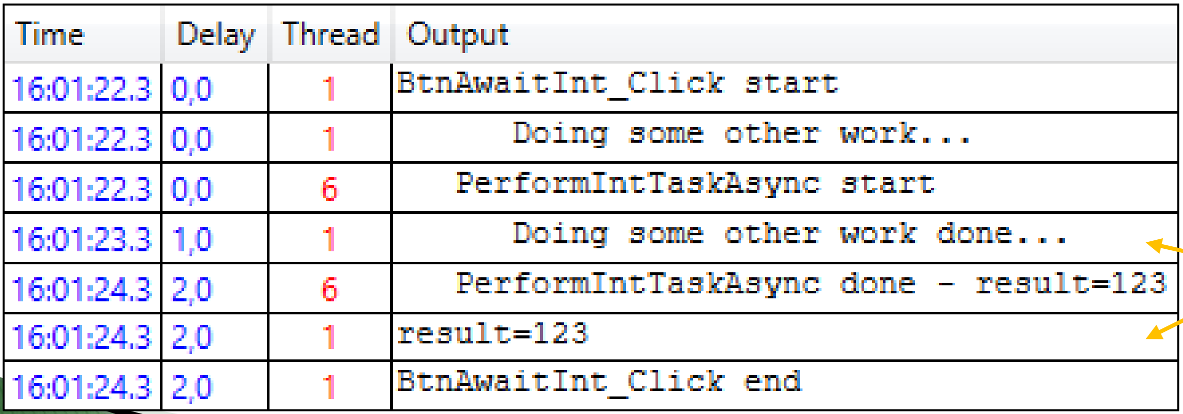


```
private async void BtnAwaitInt_Click(object sender, RoutedEventArgs e)
{
  StartLog();
  Log("BtnAwaitInt_Click start");
  var task = PerformIntTaskAsync(true);
  Log("      Doing some other work...");
  await Task.Delay(1000);
  Log("      Doing some other work done...");
  int result = await task;
  Log($"result={result}");
  Log("BtnAwaitInt_Click end");
}
```

async ist verpflichtend: "does the magic"

anstelle von Thread.Sleep()

anstelle von task.Result Blockiert nicht!

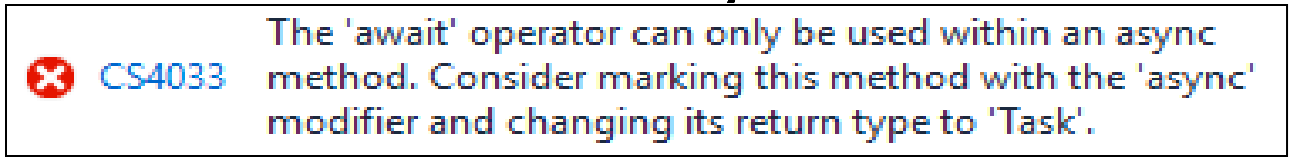


Time	Delay	Thread	Output
16:01:22.3	0,0	1	BtnAwaitInt_Click start
16:01:22.3	0,0	1	Doing some other work...
16:01:22.3	0,0	6	PerformIntTaskAsync start
16:01:23.3	1,0	1	Doing some other work done...
16:01:24.3	2,0	6	PerformIntTaskAsync done - result=123
16:01:24.3	2,0	1	result=123
16:01:24.3	2,0	1	BtnAwaitInt_Click end

Continuations im UI-Thread

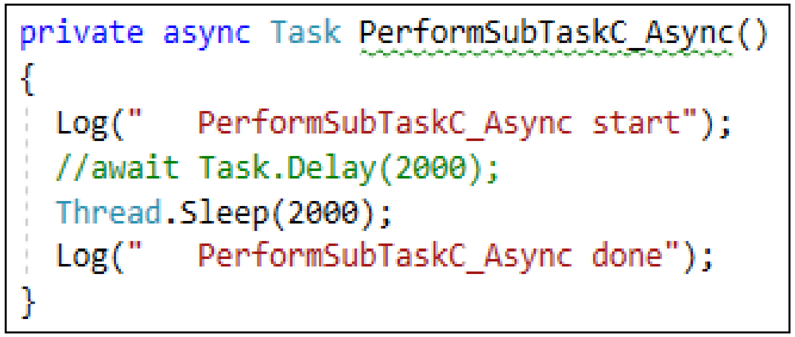
Compiler errors/warnings

- Error: await ohne async

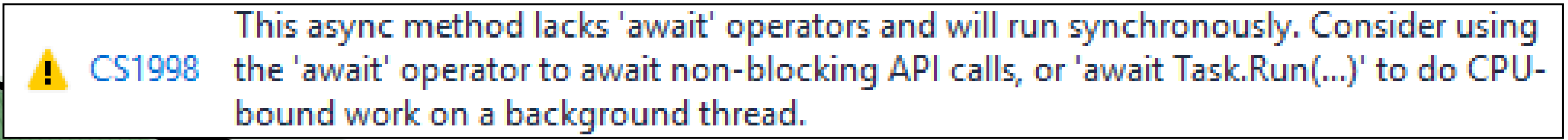


```
CS4033  The 'await' operator can only be used within an async method. Consider marking this method with the 'async' modifier and changing its return type to 'Task'.
```

- Warning: async ohne await ➔ synchron
  - Microsoft-Doku: „If an async method doesn't use an **await** operator to mark a suspension point, the method executes as a synchronous method does, despite the **async** modifier."



```
private async Task PerformSubTaskC_Async()
{
  Log("   PerformSubTaskC_Async start");
  //await Task.Delay(2000);
  Thread.Sleep(2000);
  Log("   PerformSubTaskC_Async done");
}
```



```
CS1998  This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread.
```

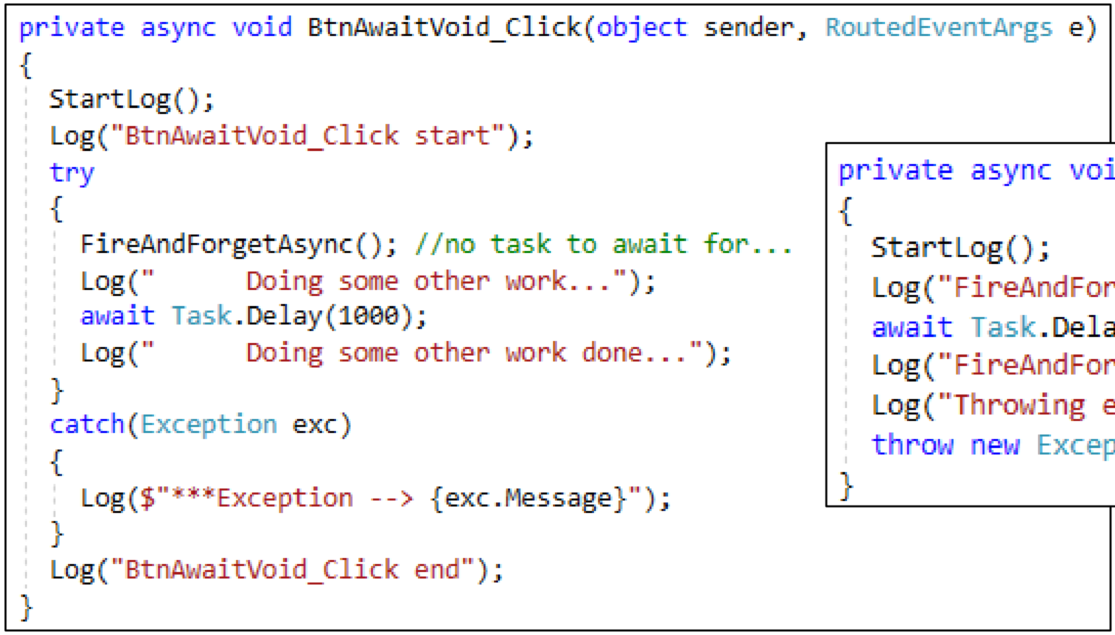
async/await Regeln

- await nur in async-Methoden möglich
- await geht nur mit awaitable, also Task u. Task<T>
- Returntyp einer async Methode wird in einen Task "eingepackt"
- Task ist Referenz auf Ergebnis oder Error
- await "entpackt" den Wert aus dem Task
- Task.Run startet einen Task über Threadpool
- await wartet auf Beendingung des Task ohne UI-Thread zu blockieren
  - task.Result/task.Wait() blockiert!
- async/await verwendet state-machine ➔nach await ist man wieder im UI-Thread
  - Dispatcher.Invoke nicht mehr notwendig

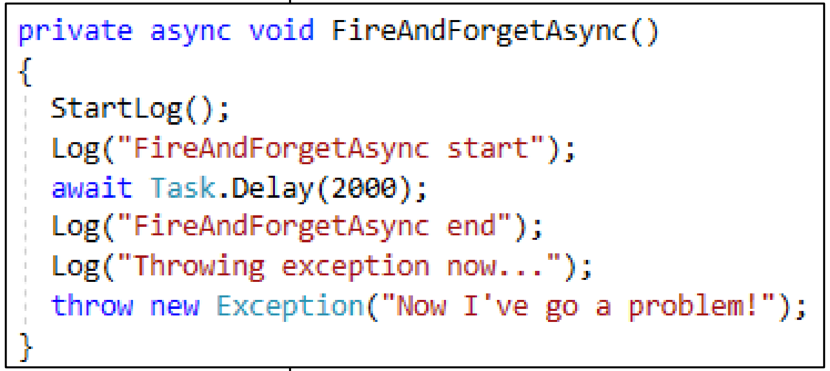
Absolutes No-Go: async void

- void ist kein awaitable ➔ kein await möglich
- Entspricht "fire and forget"
  - Beendigung der Methode kann nicht erkannt werden
  - Exceptions gehen verloren (Aufrufer ist längst weitergelaufen)
- Ausnahme: async Eventhandler
  - Signatur ist vorgegeben

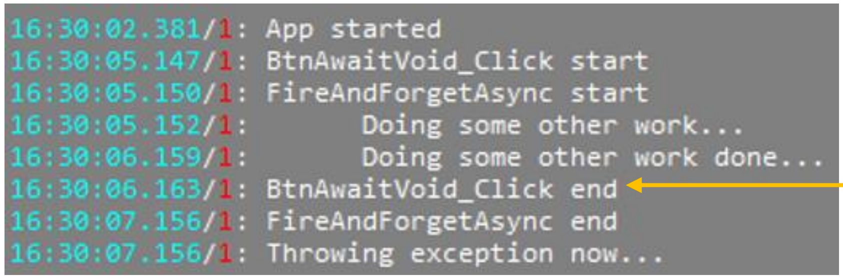
async void Beispiel



```
private async void BtnAwaitVoid_Click(object sender, RoutedEventArgs e)
{
  StartLog();
  Log("BtnAwaitVoid_Click start");
  try
  {
    FireAndForgetAsync(); //no task to await for...
    Log("       Doing some other work...");
    await Task.Delay(1000);
    Log("       Doing some other work done...");
  }
  catch(Exception exc)
  {
    Log($"***Exception --> {exc.Message}");
  }
  Log("BtnAwaitVoid_Click end");
}
```



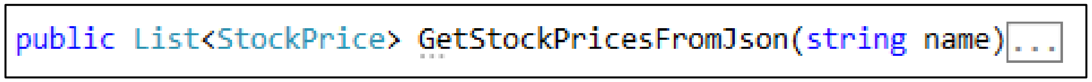
```
private async void FireAndForgetAsync()
{
  StartLog();
  Log("FireAndForgetAsync start");
  await Task.Delay(2000);
  Log("FireAndForgetAsync end");
  Log("Throwing exception now...");
  throw new Exception("Now I've go a problem!");
}
```



```
16:30:02.381/1: App started
16:30:05.147/1: BtnAwaitVoid_Click start
16:30:05.150/1: FireAndForgetAsync start
16:30:05.152/1:        Doing some other work...
16:30:06.159/1:        Doing some other work done...
16:30:06.163/1: BtnAwaitVoid_Click end
16:30:07.156/1: FireAndForgetAsync end
16:30:07.156/1: Throwing exception now...
```

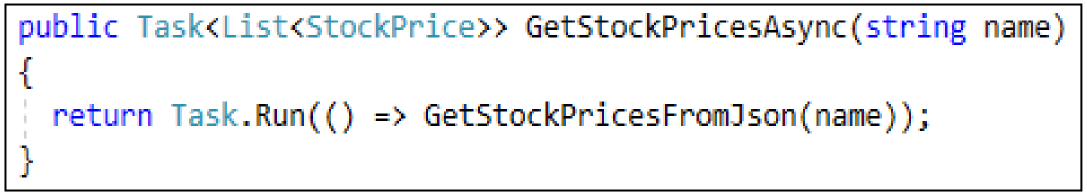
Methode ist beendet

Methode umbauen



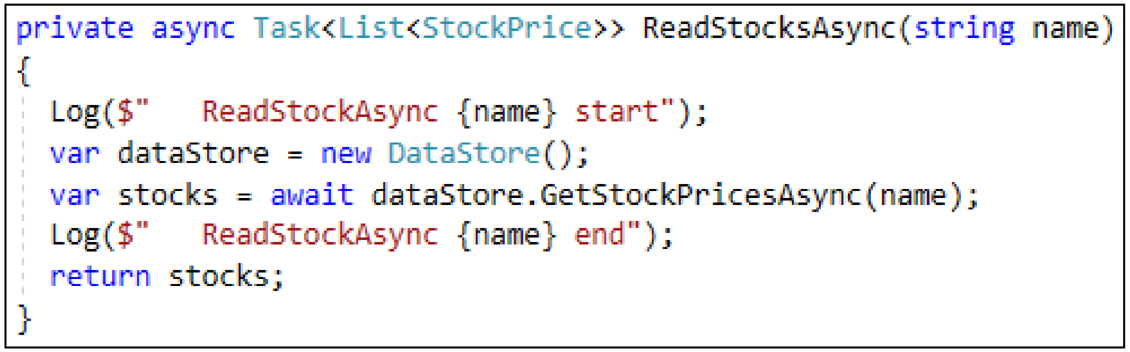
```
public List<StockPrice> GetStockPricesFromJson(string name)...
```

lang dauernde Methode



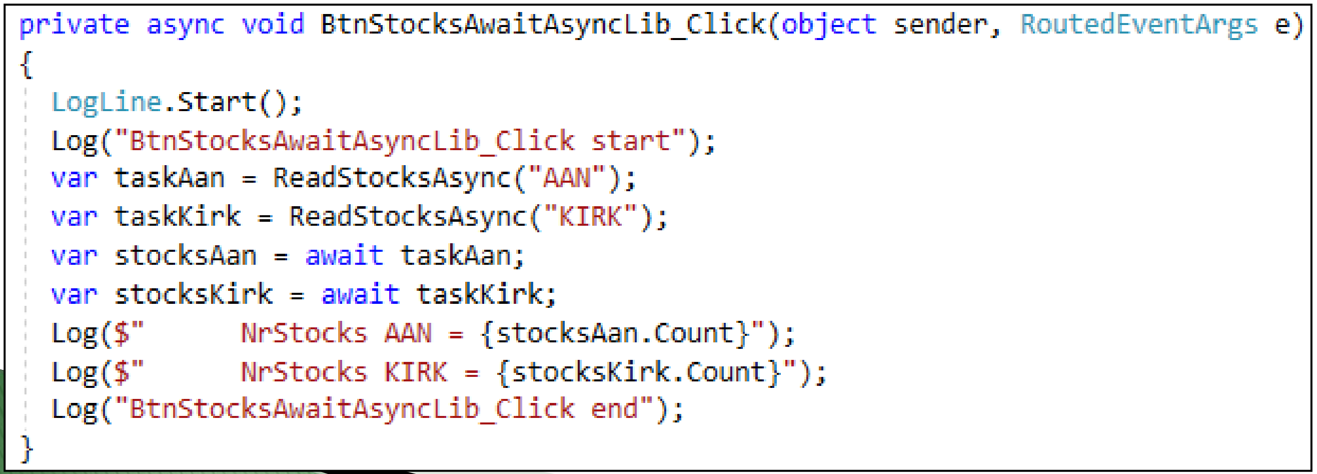
```
public Task<List<StockPrice>> GetStockPricesAsync(string name)
{
  return Task.Run(() => GetStockPricesFromJson(name));
}
```

einpacken in Task



```
private async Task<List<StockPrice>> ReadStocksAsync(string name)
{
  Log($"   ReadStockAsync {name} start");
  var dataStore = new DataStore();
  var stocks = await dataStore.GetStockPricesAsync(name);
  Log($"   ReadStockAsync {name} end");
  return stocks;
}
```

aufrufen in async Methode



```
private async void BtnStocksAwaitAsyncLib_Click(object sender, RoutedEventArgs e)
{
  LogLine.Start();
  Log("BtnStocksAwaitAsyncLib_Click start");
  var taskAan = ReadStocksAsync("AAN");
  var taskKirk = ReadStocksAsync("KIRK");
  var stocksAan = await taskAan;
  var stocksKirk = await taskKirk;
  Log($"      NrStocks AAN = {stocksAan.Count}");
  Log($"      NrStocks KIRK = {stocksKirk.Count}");
  Log("BtnStocksAwaitAsyncLib_Click end");
}
```

async/await für alle Aufrufer durchziehen

Zusammenfassung

Dos

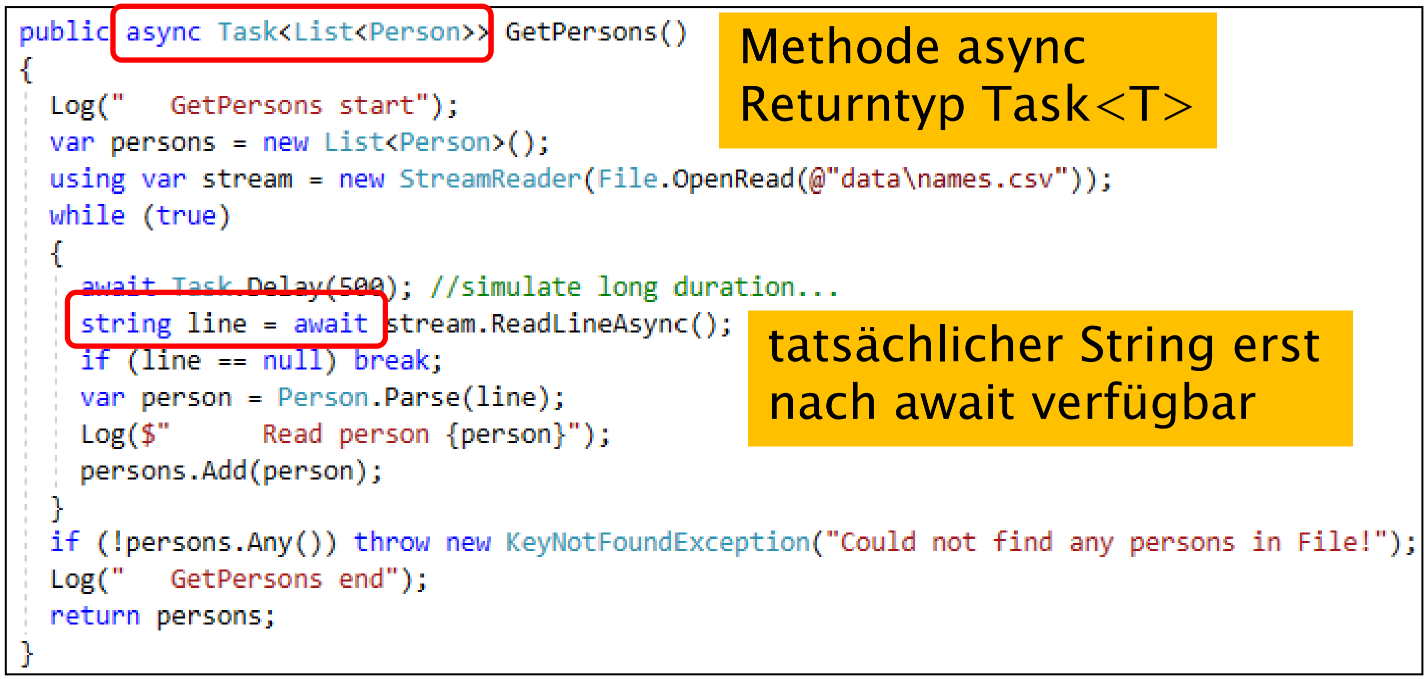
- ✓ async und await immer gemeinsam benutzen
- ✓ Asynchrone Methoden immer Task retournieren lassen
- ✓ Asynchrone Methoden immer mit await validieren
- ✓ async/await die gesamte Aufrufkette verwenden

Don'ts

- Niemals async void (außer bei Eventhandler)
- Asynchrone Methoden niemals durch Result bzw. Wait() blockieren

Third Party 1/2

- Viele Libraries sind async
- Methodennamen meist Suffix Async
- z.B. File einlesen mit ReadLineAsync



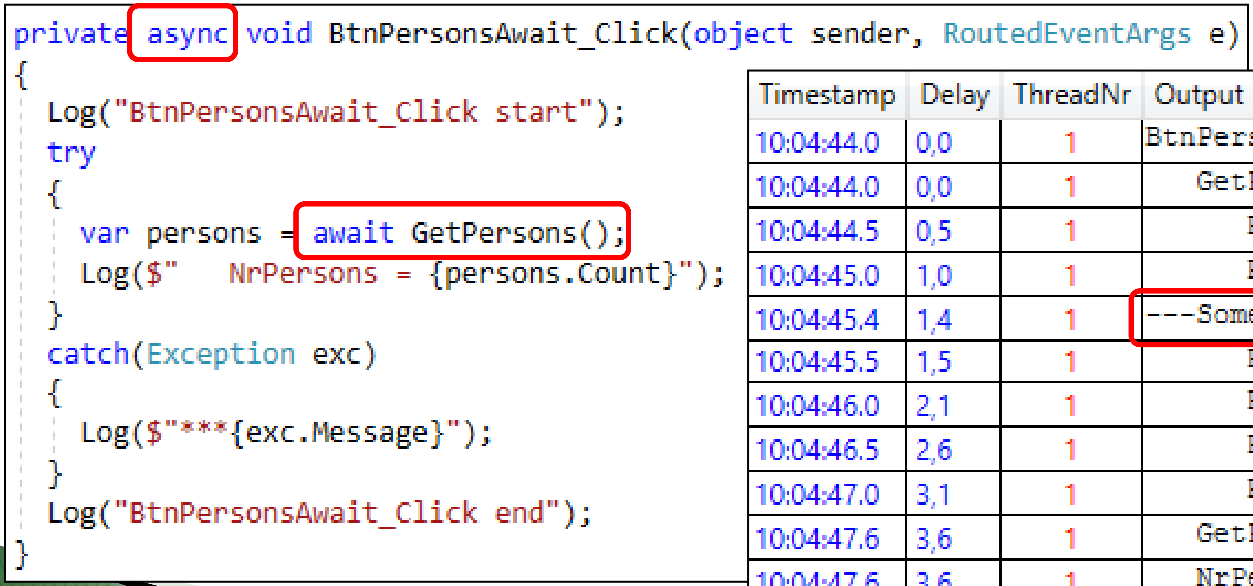
```
public async Task<List<Person>> GetPersons()
{
  Log("   GetPersons start");
  var persons = new List<Person>();
  using var stream = new StreamReader(File.OpenRead(@"data\names.csv"));
  while (true)
  {
    await Task.Delay(500); //simulate long duration...
    string line = await stream.ReadLineAsync();
    if (line == null) break;
    var person = Person.Parse(line);
    Log($"      Read person {person}");
    persons.Add(person);
  }
  if (!persons.Any()) throw new KeyNotFoundException("Could not find any persons in File!");
  Log("   GetPersons end");
  return persons;
}
```

Methode async
Returntyp Task<T>

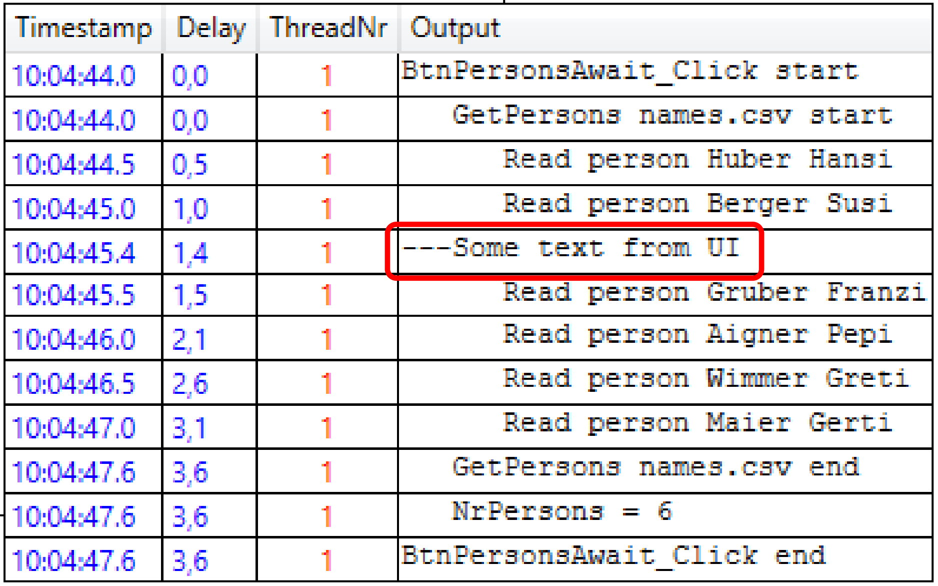
tatsächlicher String erst
nach await verfügbar

Third Party 2/2

- Beim Aufruf ebenfalls:
  - Methode async
  - Wert mit await abwarten
- App bleibt bedienbar
- Exceptions wie bei synchronem Aufruf



```
private async void BtnPersonsAwait_Click(object sender, RoutedEventArgs e)
{
  Log("BtnPersonsAwait_Click start");
  try
  {
    var persons = await GetPersons();
    Log($"   NrPersons = {persons.Count}");
  }
  catch(Exception exc)
  {
    Log($"***{exc.Message}");
  }
  Log("BtnPersonsAwait_Click end");
}
```

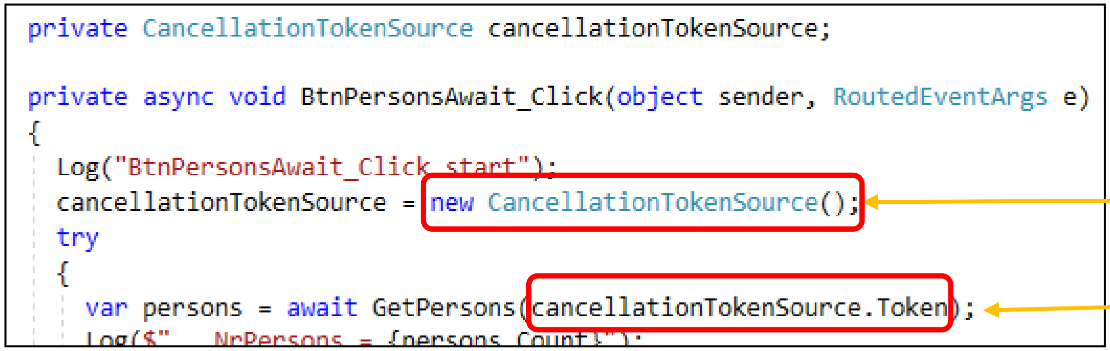


Timestamp	Delay	ThreadNr	Output
10:04:44.0	0,0	1	BtnPersonsAwait_Click start
10:04:44.0	0,0	1	GetPersons names.csv start
10:04:44.5	0,5	1	Read person Huber Hansi
10:04:45.0	1,0	1	Read person Berger Susi
10:04:45.4	1,4	1	---Some text from UI
10:04:45.5	1,5	1	Read person Gruber Franzi
10:04:46.0	2,1	1	Read person Aigner Pepi
10:04:46.5	2,6	1	Read person Wimmer Greti
10:04:47.0	3,1	1	Read person Maier Gerti
10:04:47.6	3,6	1	GetPersons names.csv end
10:04:47.6	3,6	1	NrPersons = 6
10:04:47.6	3,6	1	BtnPersonsAwait_Click end

CancellationToken 1/2

- Task vorzeitig beenden
- Ginge auch mit Flag – besser CancellationToken
- Vielen Async-Methoden in Libraries kann man ein derartiges Token mitgeben
- Vorgehensweise
  - `CancellationTokenSource` erzeugen
  - `CancellationToken` über die Property `Token` holen
  - Dieses Token dem Task übergeben
  - Im Task `IsCancellationRequested` an geeigneter Stelle prüfen
  - Task vorzeitig beenden mit der Methode `Cancel()` der CancellationTokenSource
  - Ein gecanceltes Token bleibt gecancelt ➔ immer neue CancellationTokenSource erzeugen

CancellationToken 2/2

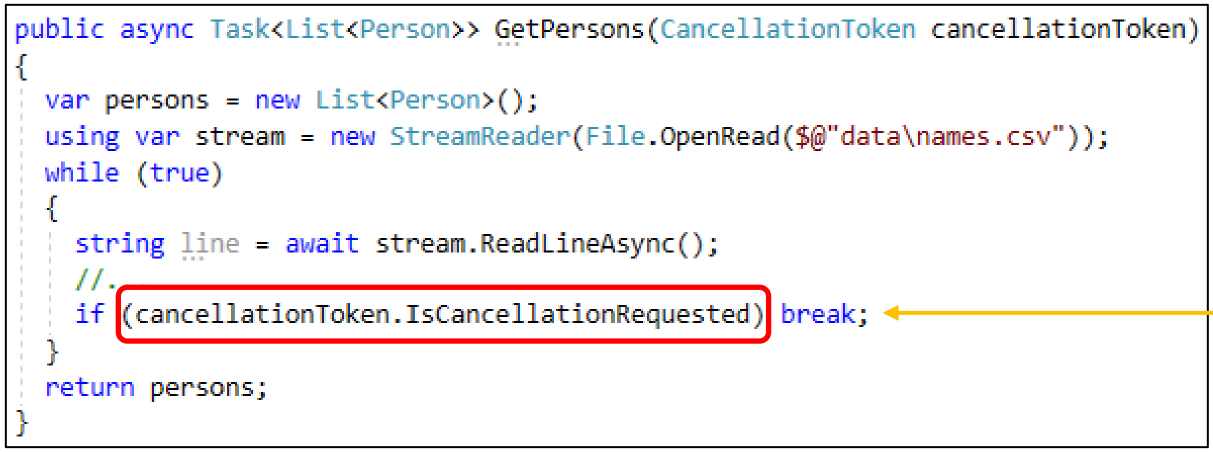


```
private CancellationTokenSource cancellationTokenSource;

private async void BtnPersonsAwait_Click(object sender, RoutedEventArgs e)
{
  Log("BtnPersonsAwait_Click start");
  cancellationTokenSource = new CancellationTokenSource();
  try
  {
    var persons = await GetPersons(cancellationTokenSource.Token);
    Log($"  NrPersons = {persons.Count}");
```

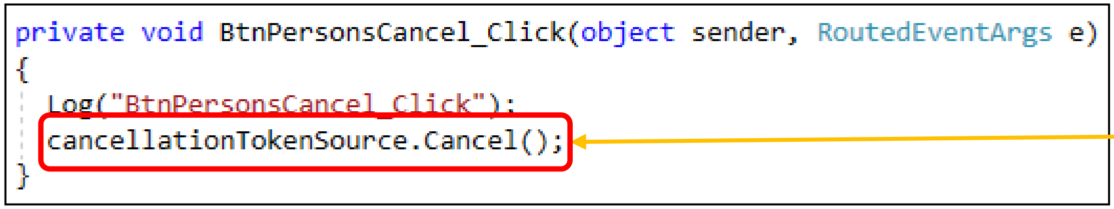
Erzeugen

Übergeben



```
public async Task<List<Person>> GetPersons(CancellationToken cancellationToken)
{
  var persons = new List<Person>();
  using var stream = new StreamReader(File.OpenRead($@"data\names.csv"));
  while (true)
  {
    string line = await stream.ReadLineAsync();
    //.
    if (cancellationToken.IsCancellationRequested) break;
  }
  return persons;
}
```

Prüfen

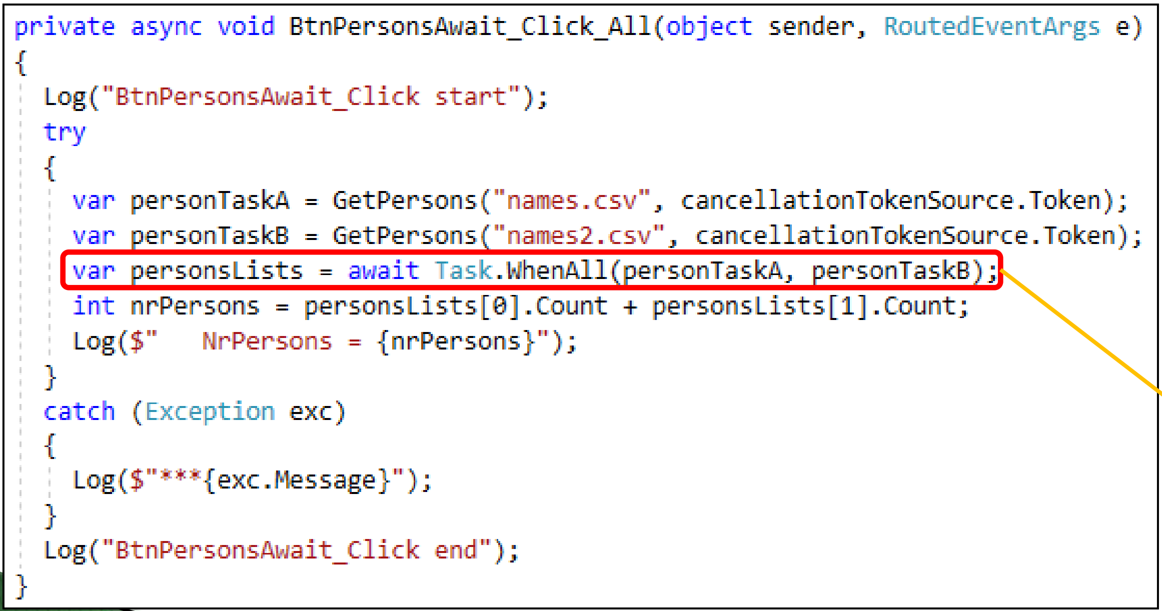


```
private void BtnPersonsCancel_Click(object sender, RoutedEventArgs e)
{
  Log("BtnPersonsCancel_Click");
  cancellationTokenSource.Cancel();
}
```

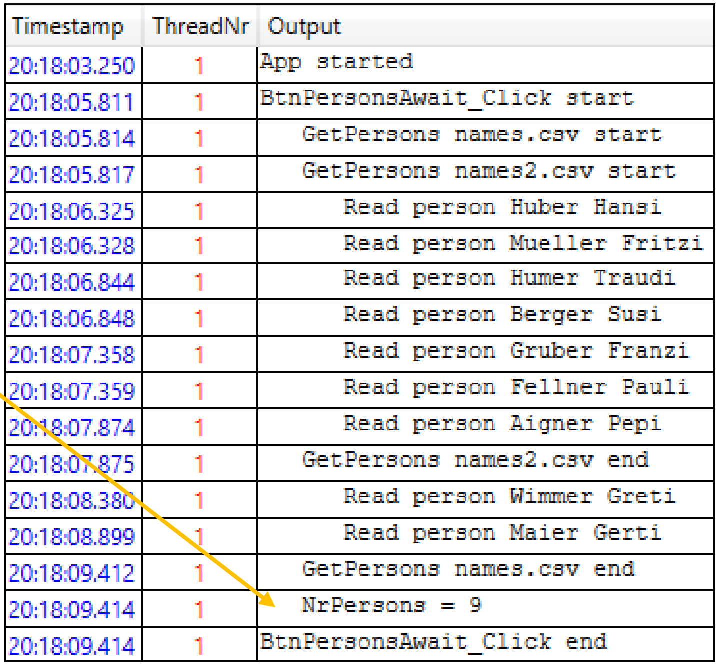
Canceln

WhenAll / WhenAny

- Mehrere Tasks kombinieren
- **`WhenAll`**: terminiert, wenn alle Tasks beendet sind
  - Ergebnisse liegen als Array vor
- **`WhenAny`**: terminiert, sobald ein Task beendet ist



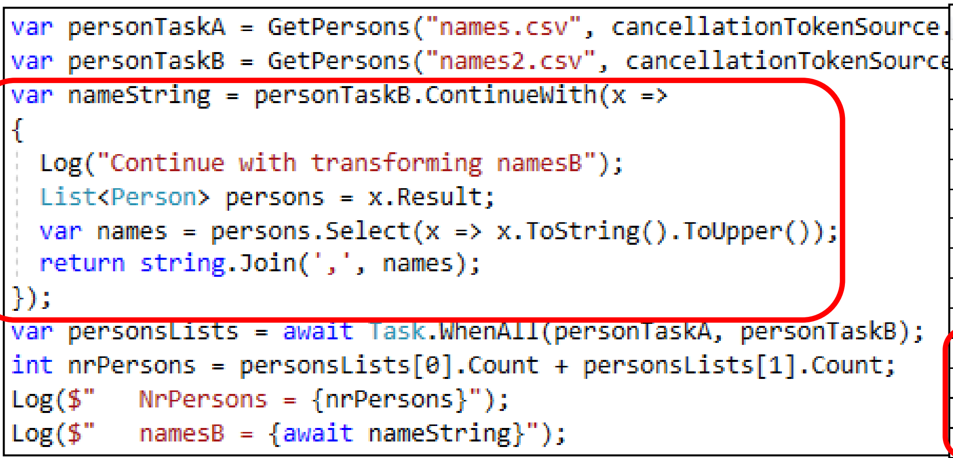
```csharp
private async void BtnPersonsAwait_Click_All(object sender, RoutedEventArgs e)
{
  Log("BtnPersonsAwait_Click start");
  try
  {
    var personTaskA = GetPersons("names.csv", cancellationTokenSource.Token);
    var personTaskB = GetPersons("names2.csv", cancellationTokenSource.Token);
    var personsLists = await Task.WhenAll(personTaskA, personTaskB);
    int nrPersons = personsLists[0].Count + personsLists[1].Count;
    Log($"   NrPersons = {nrPersons}");
  }
  catch (Exception exc)
  {
    Log($"***{exc.Message}");
  }
  Log("BtnPersonsAwait_Click end");
}
```



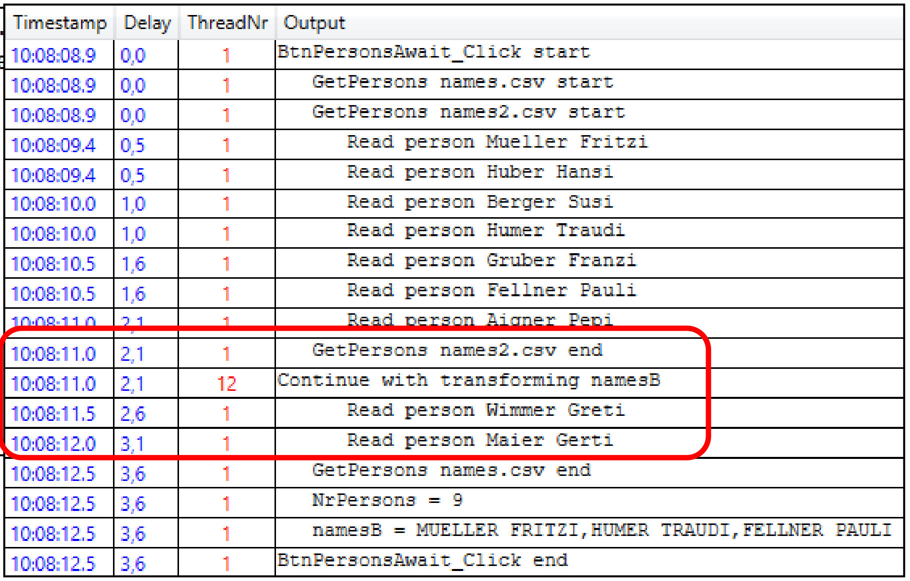
Timestamp	ThreadNr	Output
20:18:03.250	1	App started
20:18:05.811	1	BtnPersonsAwait_Click start
20:18:05.814	1	GetPersons names.csv start
20:18:05.817	1	GetPersons names2.csv start
20:18:06.325	1	Read person Huber Hansi
20:18:06.328	1	Read person Mueller Fritzi
20:18:06.844	1	Read person Humer Traudi
20:18:06.848	1	Read person Berger Susi
20:18:07.358	1	Read person Gruber Franzi
20:18:07.359	1	Read person Fellner Pauli
20:18:07.874	1	Read person Aigner Pepi
20:18:07.875	1	GetPersons names2.csv end
20:18:08.380	1	Read person Wimmer Greti
20:18:08.899	1	Read person Maier Gerti
20:18:09.412	1	GetPersons names.csv end
20:18:09.414	1	NrPersons = 9
20:18:09.414	1	BtnPersonsAwait_Click end

Continuations

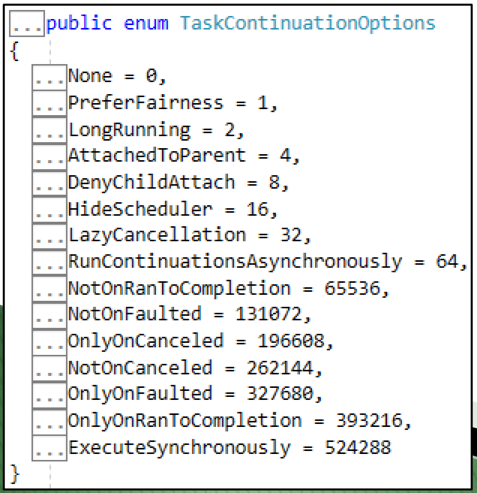
- Standard-continuation: nach await im UI-Thread
- **ContinueWith**: in weiterem Task weiterlaufen
- **TaskContinuationOptions**: nur unter bestimmten Bedingungen



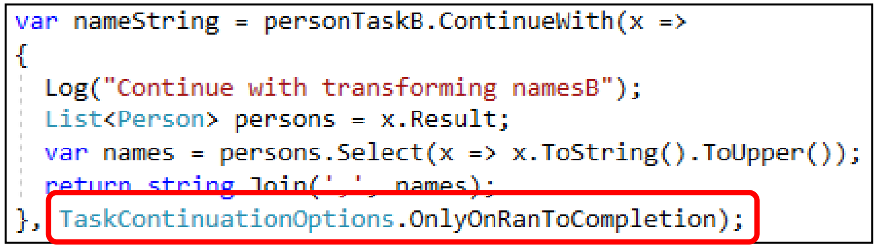
```
var personTaskA = GetPersons("names.csv", cancellationTokenSource.
var personTaskB = GetPersons("names2.csv", cancellationTokenSource
var nameString = personTaskB.ContinueWith(x =>
{
  Log("Continue with transforming namesB");
  List<Person> persons = x.Result;
  var names = persons.Select(x => x.ToString().ToUpper());
  return string.Join(',', names);
});
var personsLists = await Task.WhenAll(personTaskA, personTaskB);
int nrPersons = personsLists[0].Count + personsLists[1].Count;
Log($"   NrPersons = {nrPersons}");
Log($"   namesB = {await nameString}");
```



Timestamp	Delay	ThreadNr	Output
10:08:08.9	0,0	1	BtnPersonsAwait_Click start
10:08:08.9	0,0	1	GetPersons names.csv start
10:08:08.9	0,0	1	GetPersons names2.csv start
10:08:09.4	0,5	1	Read person Mueller Fritzi
10:08:09.4	0,5	1	Read person Huber Hansi
10:08:10.0	1,0	1	Read person Berger Susi
10:08:10.0	1,0	1	Read person Humer Traudi
10:08:10.5	1,6	1	Read person Gruber Franzi
10:08:10.5	1,6	1	Read person Fellner Pauli
10:08:11.0	2,1	1	Read person Aigner Pepi
10:08:11.0	2,1	1	GetPersons names2.csv end
10:08:11.0	2,1	12	Continue with transforming namesB
10:08:11.5	2,6	1	Read person Wimmer Greti
10:08:12.0	3,1	1	Read person Maier Gerti
10:08:12.5	3,6	1	GetPersons names.csv end
10:08:12.5	3,6	1	NrPersons = 9
10:08:12.5	3,6	1	namesB = MUELLER FRITZI,HUMER TRAUDI,FELLNER PAULI
10:08:12.5	3,6	1	BtnPersonsAwait_Click end



```
...public enum TaskContinuationOptions
{
  ...None = 0,
  ...PreferFairness = 1,
  ...LongRunning = 2,
  ...AttachedToParent = 4,
  ...DenyChildAttach = 8,
  ...HideScheduler = 16,
  ...LazyCancellation = 32,
  ...RunContinuationsAsynchronously = 64,
  ...NotOnRanToCompletion = 65536,
  ...NotOnFaulted = 131072,
  ...OnlyOnCanceled = 196608,
  ...NotOnCanceled = 262144,
  ...OnlyOnFaulted = 327680,
  ...OnlyOnRanToCompletion = 393216,
  ...ExecuteSynchronously = 524288
}
```

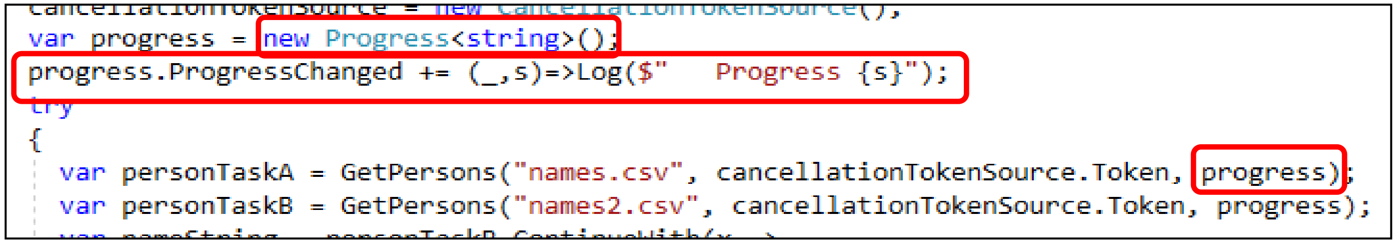


```
var nameString = personTaskB.ContinueWith(x =>
{
  Log("Continue with transforming namesB");
  List<Person> persons = x.Result;
  var names = persons.Select(x => x.ToString().ToUpper());
  return string.Join(',', names);
}, TaskContinuationOptions.OnlyOnRanToCompletion);
```

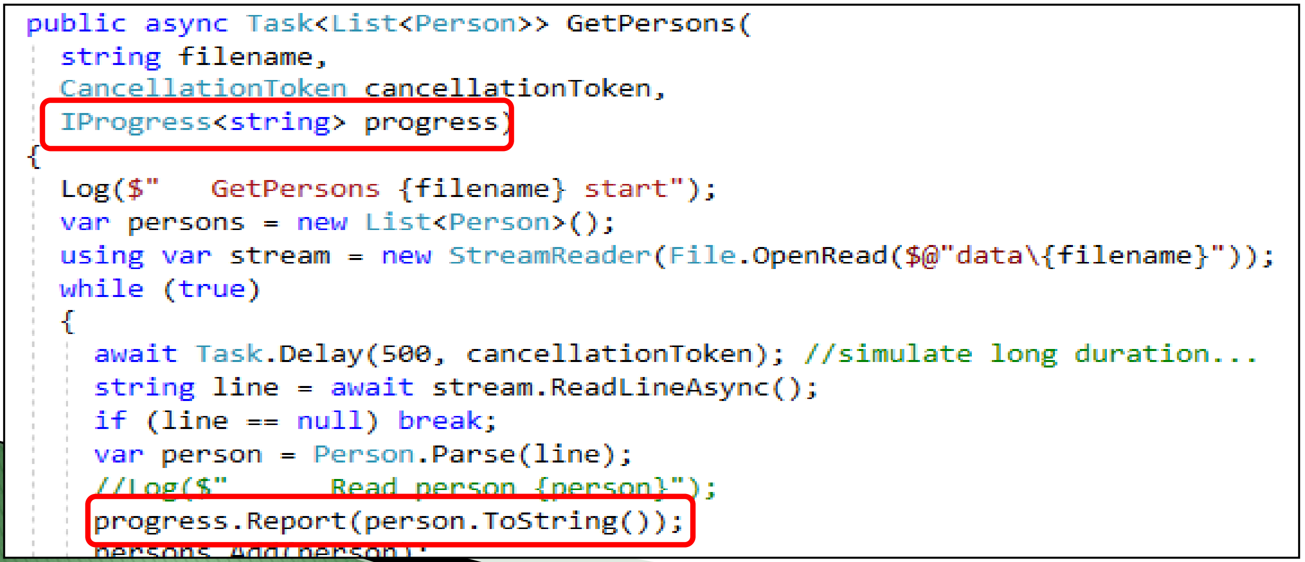
Ohne diese Option entsteht Exception bei Task-Cancel

IProgress<T>

- Fortschritts-Rückmeldung an Aufrufer
- **Progress-Objekt** übergeben
- Fortschritt verarbeiten: Event **ProgressChanged**
- Rückmelden über Methode **Report**



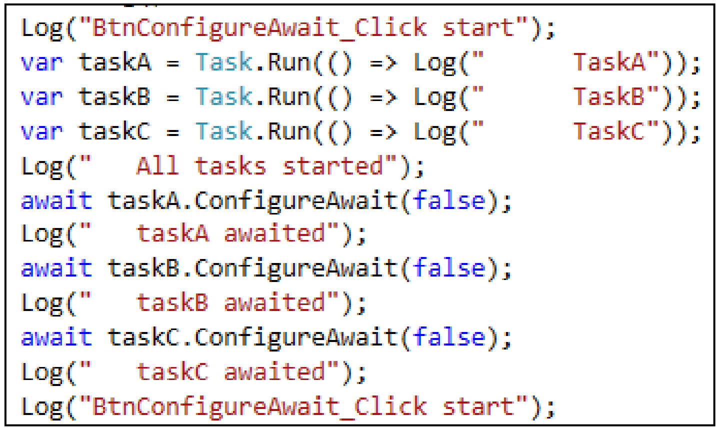
```
cancellationTokenSource = new CancellationTokenSource();
var progress = new Progress<string>();
progress.ProgressChanged += (_,s)=>Log($"    Progress {s}");
try
{
  var personTaskA = GetPersons("names.csv", cancellationTokenSource.Token, progress);
  var personTaskB = GetPersons("names2.csv", cancellationTokenSource.Token, progress);
  var nameString = personTaskB.ContinueWith(x =>
```



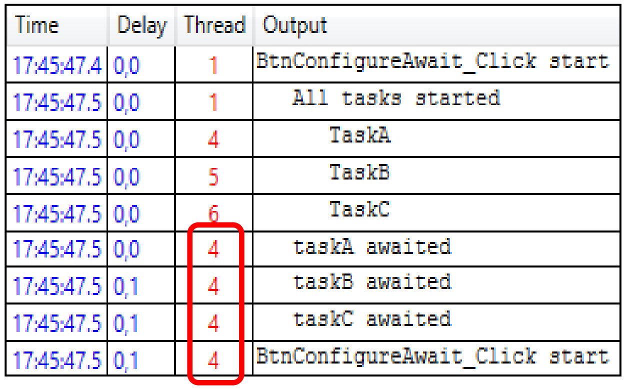
```
public async Task<List<Person>> GetPersons(
  string filename,
  CancellationToken cancellationToken,
  IProgress<string> progress)
{
  Log($"    GetPersons {filename} start");
  var persons = new List<Person>();
  using var stream = new StreamReader(File.OpenRead($@"data\{filename}"));
  while (true)
  {
    await Task.Delay(500, cancellationToken); //simulate long duration...
    string line = await stream.ReadLineAsync();
    if (line == null) break;
    var person = Person.Parse(line);
    //Log($"        Read person {person}");
    progress.Report(person.ToString());
    persons.Add(person);
```

Continuations

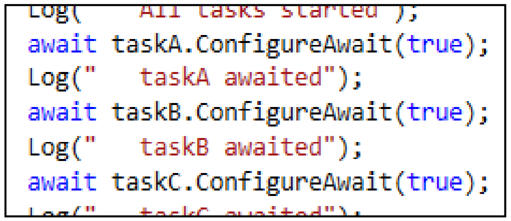
- Standard-continuation: nach await im UI-Thread
- Verhindern durch **ConfigureAwait**(false)



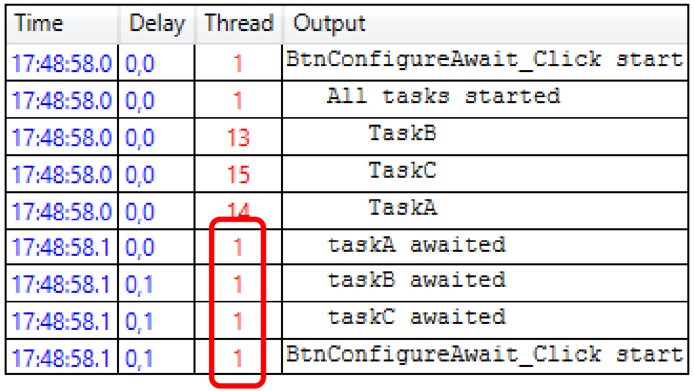
```
Log("BtnConfigureAwait_Click start");
var taskA = Task.Run(() => Log("      TaskA"));
var taskB = Task.Run(() => Log("      TaskB"));
var taskC = Task.Run(() => Log("      TaskC"));
Log("   All tasks started");
await taskA.ConfigureAwait(false);
Log("   taskA awaited");
await taskB.ConfigureAwait(false);
Log("   taskB awaited");
await taskC.ConfigureAwait(false);
Log("   taskC awaited");
Log("BtnConfigureAwait_Click start");
```



Time	Delay	Thread	Output
17:45:47.4	0,0	1	BtnConfigureAwait_Click start
17:45:47.5	0,0	1	All tasks started
17:45:47.5	0,0	4	TaskA
17:45:47.5	0,0	5	TaskB
17:45:47.5	0,0	6	TaskC
17:45:47.5	0,0	4	taskA awaited
17:45:47.5	0,1	4	taskB awaited
17:45:47.5	0,1	4	taskC awaited
17:45:47.5	0,1	4	BtnConfigureAwait_Click start



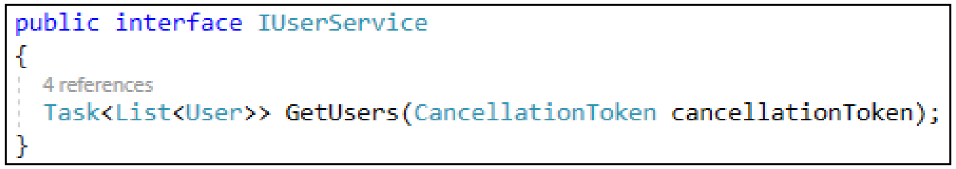
```
Log("   All tasks started");
await taskA.ConfigureAwait(true);
Log("   taskA awaited");
await taskB.ConfigureAwait(true);
Log("   taskB awaited");
await taskC.ConfigureAwait(true);
Log("   taskC awaited");
```



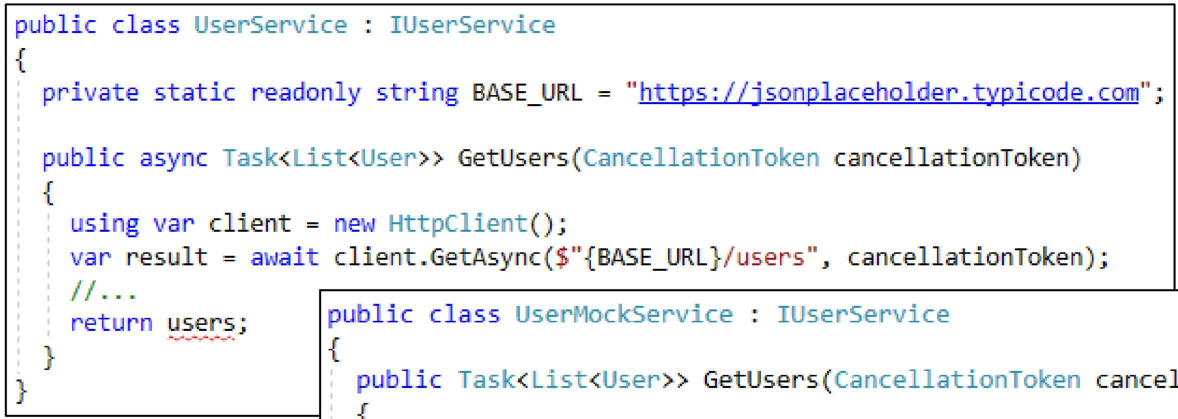
Time	Delay	Thread	Output
17:48:58.0	0,0	1	BtnConfigureAwait_Click start
17:48:58.0	0,0	1	All tasks started
17:48:58.0	0,0	13	TaskB
17:48:58.0	0,0	15	TaskC
17:48:58.0	0,0	14	TaskA
17:48:58.1	0,0	1	taskA awaited
17:48:58.1	0,1	1	taskB awaited
17:48:58.1	0,1	1	taskC awaited
17:48:58.1	0,1	1	BtnConfigureAwait_Click start

FromResult

- Fixe Werte zurückgeben, aber Schnittstelle nicht ändern
- Vor allem bei UnitTests interessant (MockService)

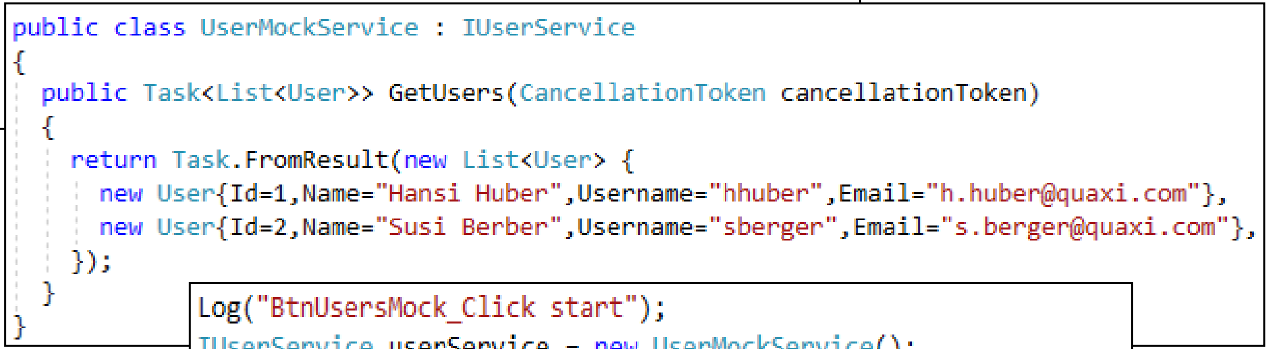


```
public interface IUserService
{
  4 references
  Task<List<User>> GetUsers(CancellationToken cancellationToken);
}
```

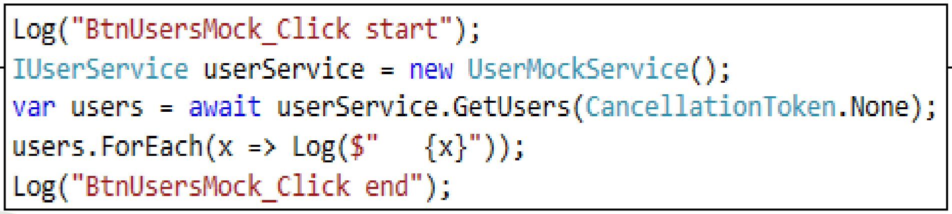


```
public class UserService : IUserService
{
  private static readonly string BASE_URL = "https://jsonplaceholder.typicode.com";

  public async Task<List<User>> GetUsers(CancellationToken cancellationToken)
  {
    using var client = new HttpClient();
    var result = await client.GetAsync($"{BASE_URL}/users", cancellationToken);
    //...
    return users;
  }
}
```



```
public class UserMockService : IUserService
{
  public Task<List<User>> GetUsers(CancellationToken cancellationToken)
  {
    return Task.FromResult(new List<User> {
      new User{Id=1,Name="Hansi Huber",Username="hhuber",Email="h.huber@quaxi.com"},
      new User{Id=2,Name="Susi Berber",Username="sberger",Email="s.berger@quaxi.com"},
    });
  }
}
```



```
Log("BtnUsersMock_Click start");
IUserService userService = new UserMockService();
var users = await userService.GetUsers(CancellationToken.None);
users.ForEach(x => Log($"    {x}"));
Log("BtnUsersMock_Click end");
```