Async und Await – Asynchrone Programmierung in C# 5

Prof. Dr. Luc Bläser

Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

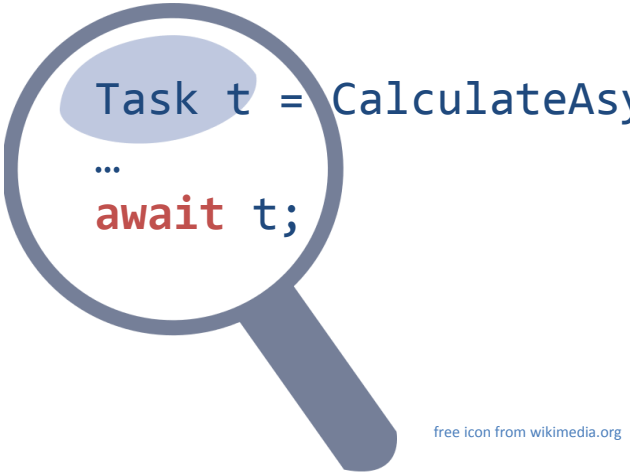
INSTITUT
FÜR
SOFTWARE

Parallel 2014, Karlsruhe
06. Mai 2014

Ausgangslage

- Async/Await Sprachkonstrukte
  - Institutionalisiert in C# 5 (und VB.NET)
  - Basierend auf Task Parallel Library (TPL) in .NET 4.5/WinRT
- First-Class asynchrones Programmiermodell
  - Substanziell anders als gewöhnliche asynchrone Aufrufe
  - Diverse Besonderheiten und Fallstricke

```
async Task CalculateAsync() {
    …
}
```



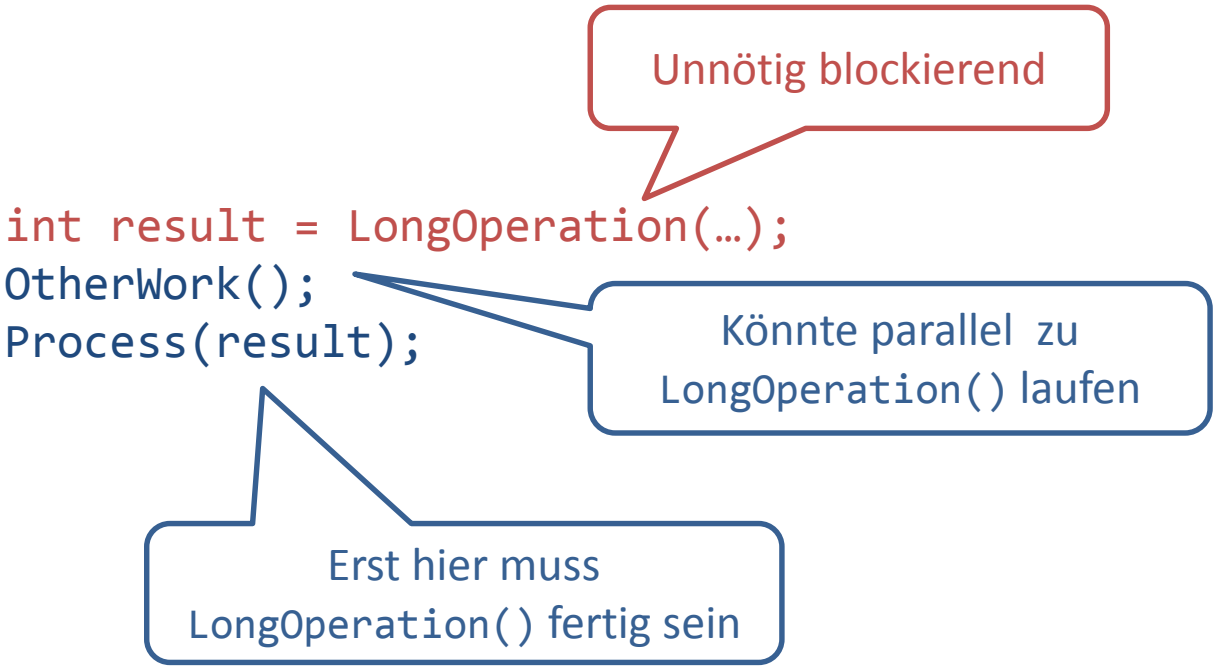
free icon from wikimedia.org

Überblick

- Async/Await Funktionsweise
- Ausführungsmodell
- Fallstricke
- Schlussfolgerung

Asynchrone Programmierung: Zweck

- Methoden nebenläufig zum Aufrufer ausführen
  - Aufrufer soll nicht unnötig blockiert werden
  - Bei I/O Calls, Service Aufrufe, Berechnungen

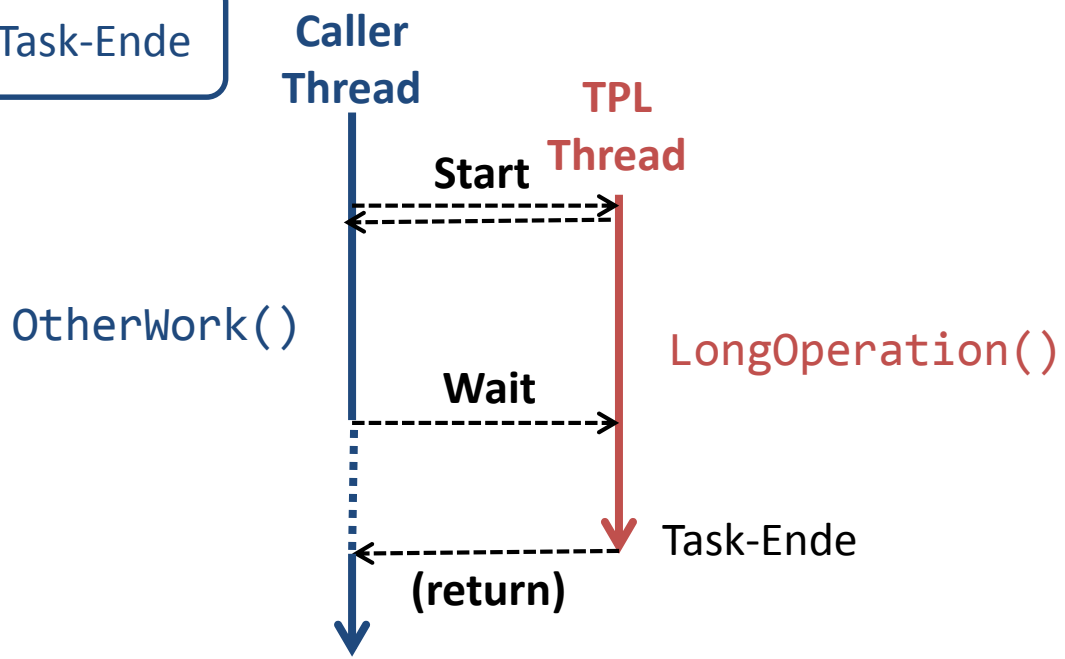


Klassische Asynchronität mit TPL

```
…
Task<int> task =
Task.Factory.StartNew<int>(LongOperation);
OtherWork();
int result = task.Result;
…
```

In Task auslagern

Warte auf Task-Ende



Alternative: Task ruft «Completion Callback»/Folgearbeit nach Beendigung selber auf

Asynchronität mit Async/Await

Async Methode

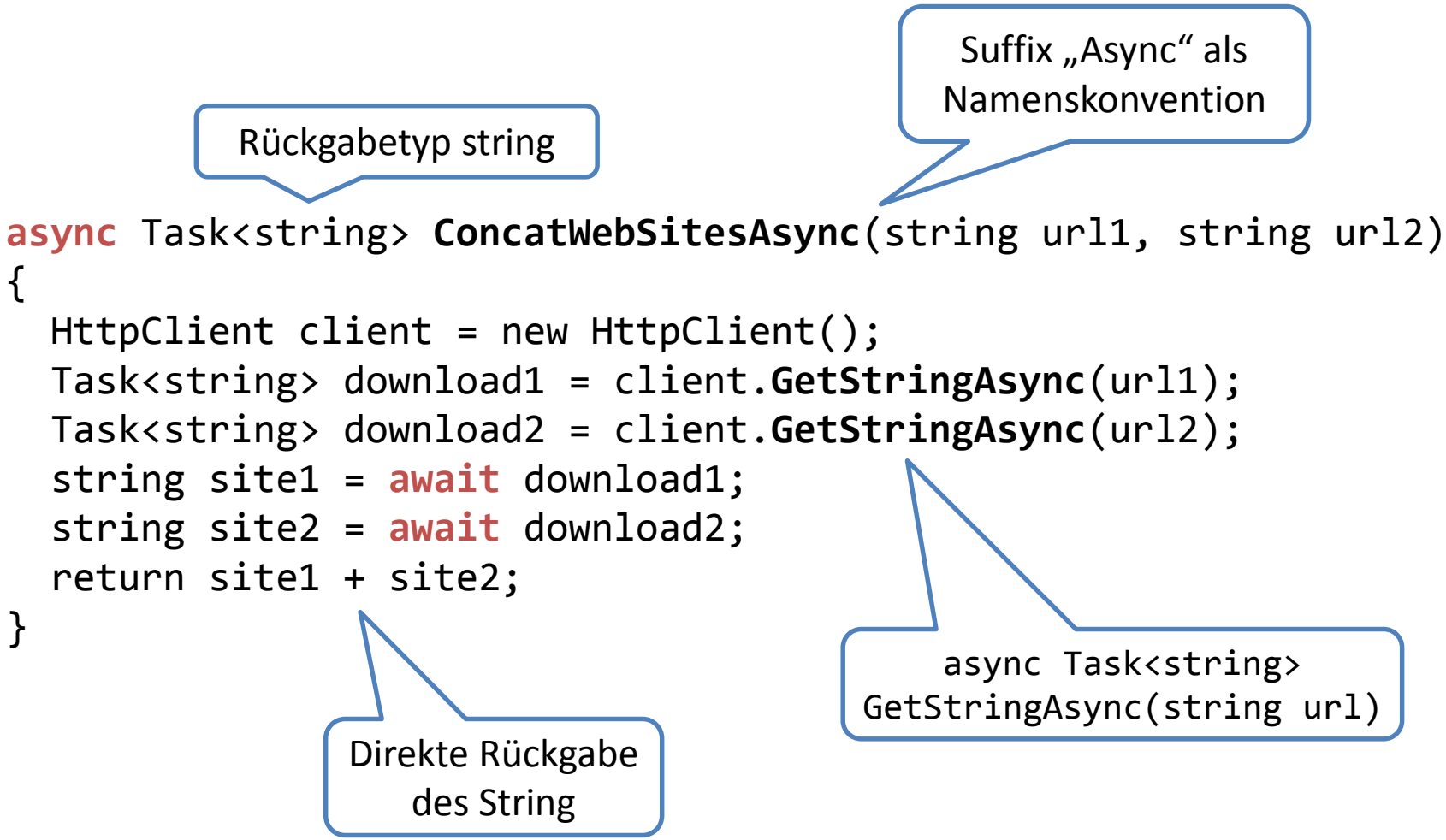
```
public async Task<int> LongOperationAsync() { … }

…
Task<int> task = LongOperationAsync();
OtherWork();
int result = await task;
…
```

Warte auf Beendigung der Async Methode

Async und Await

- Schlüsselwort async für Methode
  - Aufrufer ist nicht zwingend während der gesamten Ausführung der Async Methode blockiert
  - Mögliche Rückgabetypen
    - `void`: «fire-and-forget»
    - `Task`: Keine Rückgabe, aber erlaubt Warten auf Ende
    - `Task<T>`: Für Methode mit Rückgabetyp T
  - Keine `ref` oder `out` Parameter
- Schlüsselwort await für Tasks
  - Warte auf Ende eines TPL Tasks (sog. „Awaiter“)
  - Liefert Resultat des Tasks (sofern Rückgabetyp definiert)

Beispiel: Async Methode



Spezielle Regeln

- async Methode
  - Muss await enthalten => Sonst Compiler-Warnung
- await Anweisung
  - Nur in async Methode erlaubt => Sonst Compiler-Fehler

Grund: Spezielles Ausführungsmodell

Ausführungsmodell

- Async Methode läuft teilweise synchron, teilweise asynchron
  - Aufrufer führt Methode solange synchron aus, bis ein blockierendes await anliegt
  - Danach läuft die Methode asynchron

```
async Task<int> GetSiteLengthAsync(string url) {
  HttpClient client = new HttpClient();
  Task<string> task = client.GetStringAsync(url);
  string site1 = await task;
  return site1.Length;
}
```

Synchron
(Aufrufer Thread)

Asynchron
(evtl. anderer Thread)

Mechanismus

- Compiler zerlegt Methode in Abschnitte
  - Erster Abschnitt vor Await: Synchron durch Aufrufer
  - Abschnitt nach Await: Läuft später nach Task-Ende («Continuation»)

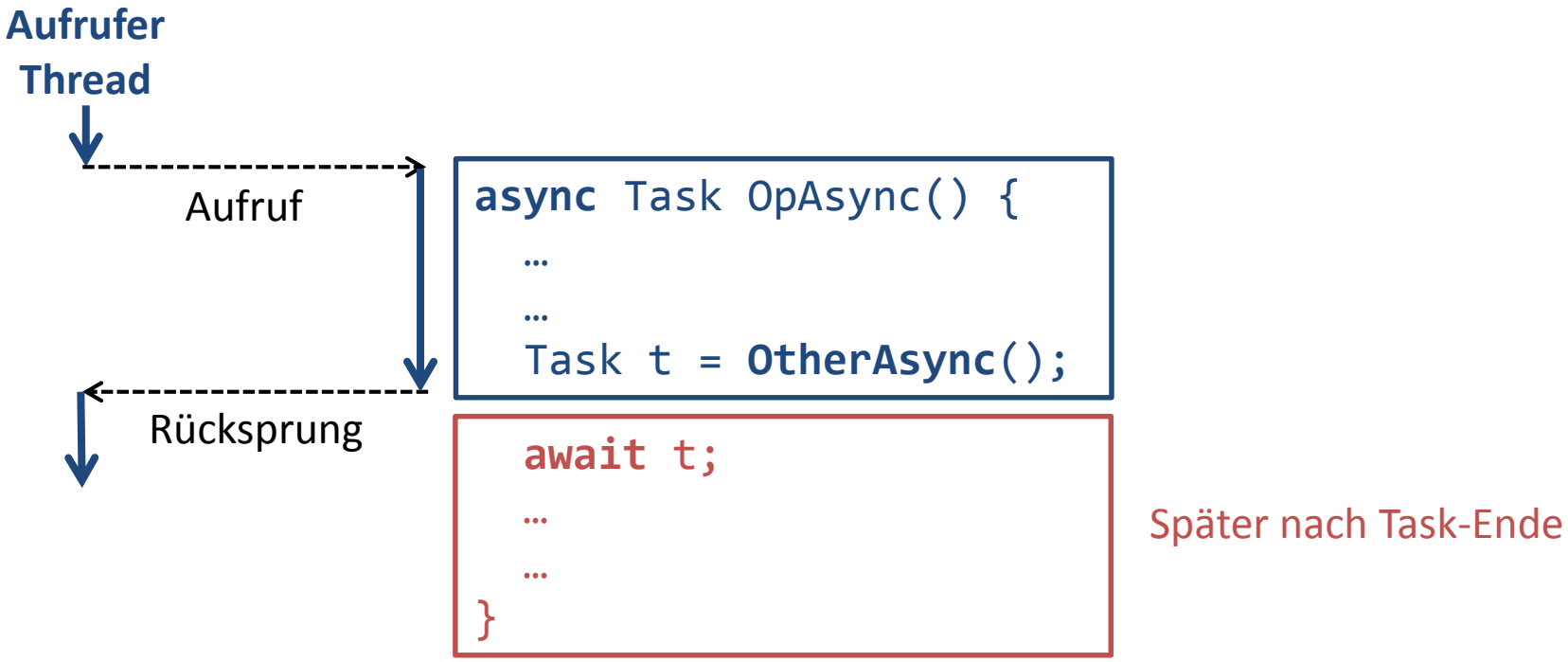
Synchron durch Aufrufer

```
async Task OpAsync() {
  …
  …
  Task t = OtherAsync();
  await t;
  …
  …
}
```

Asynchron nach Task-Ende

Async Methodenaufruf

- Methode läuft synchron bis zu blockierenden Await
  - Bei Warten auf anderen Thread bzw. externes I/O
- Bei blockierendem Await Rücksprung zum Aufrufer

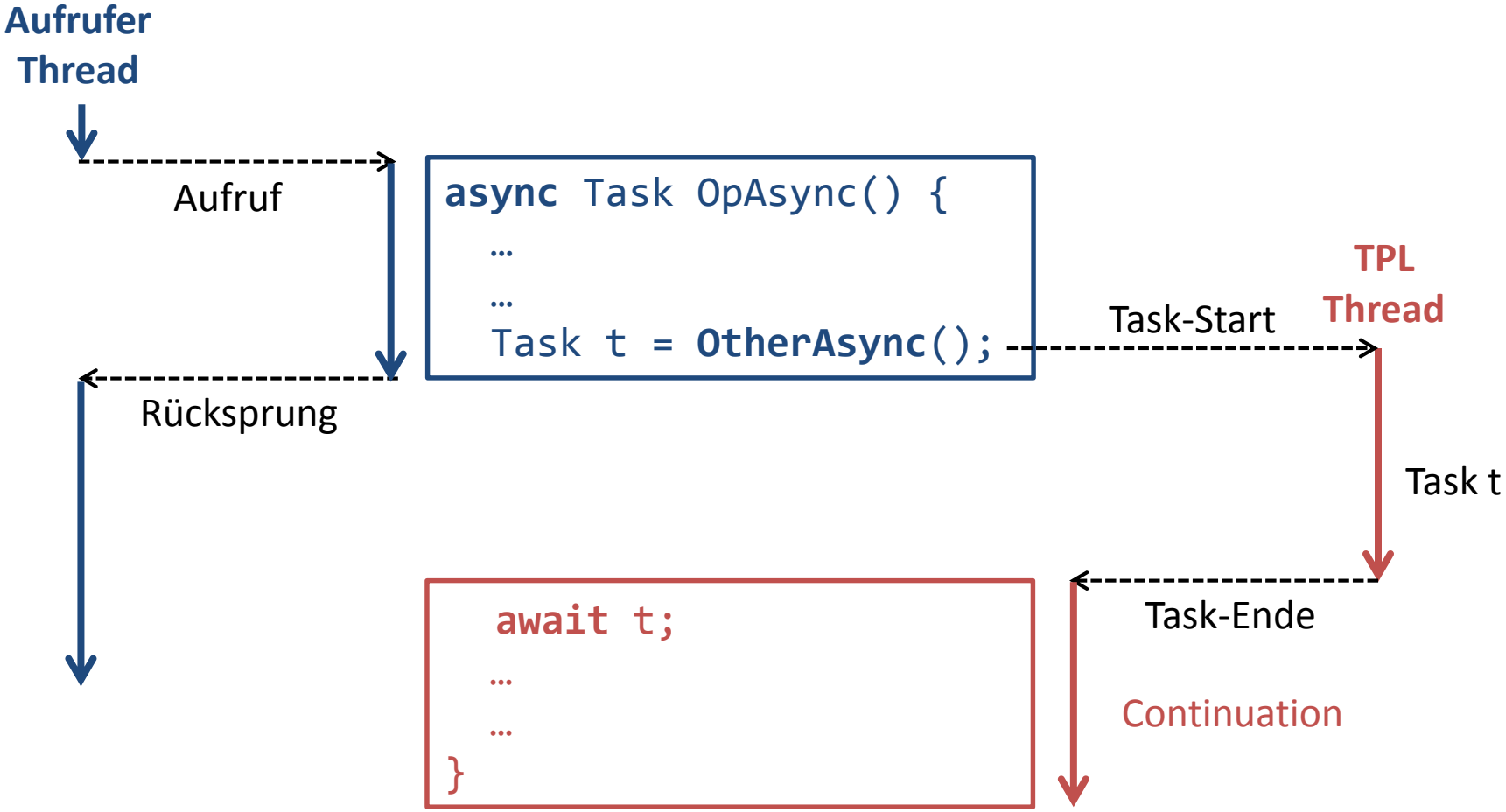


Abschnitt nach blockierendem Await

- Fall 1: Aufrufer ohne Synchronisationskontext
  - `SynchronizationContext.Current == null`
    - Meiste Threads (Konsole, TPL etc.)
  - Abschnitt wird durch Thread des erwarteten Tasks ausgeführt
- Fall 2: Aufrufer mit Synchronisationskontext
  - Z.B. GUI-Thread
  - Abschnitt wird an diesen Kontext «dispatched»
    - Wird z.B. als Event vom GUI-Thread ausgeführt

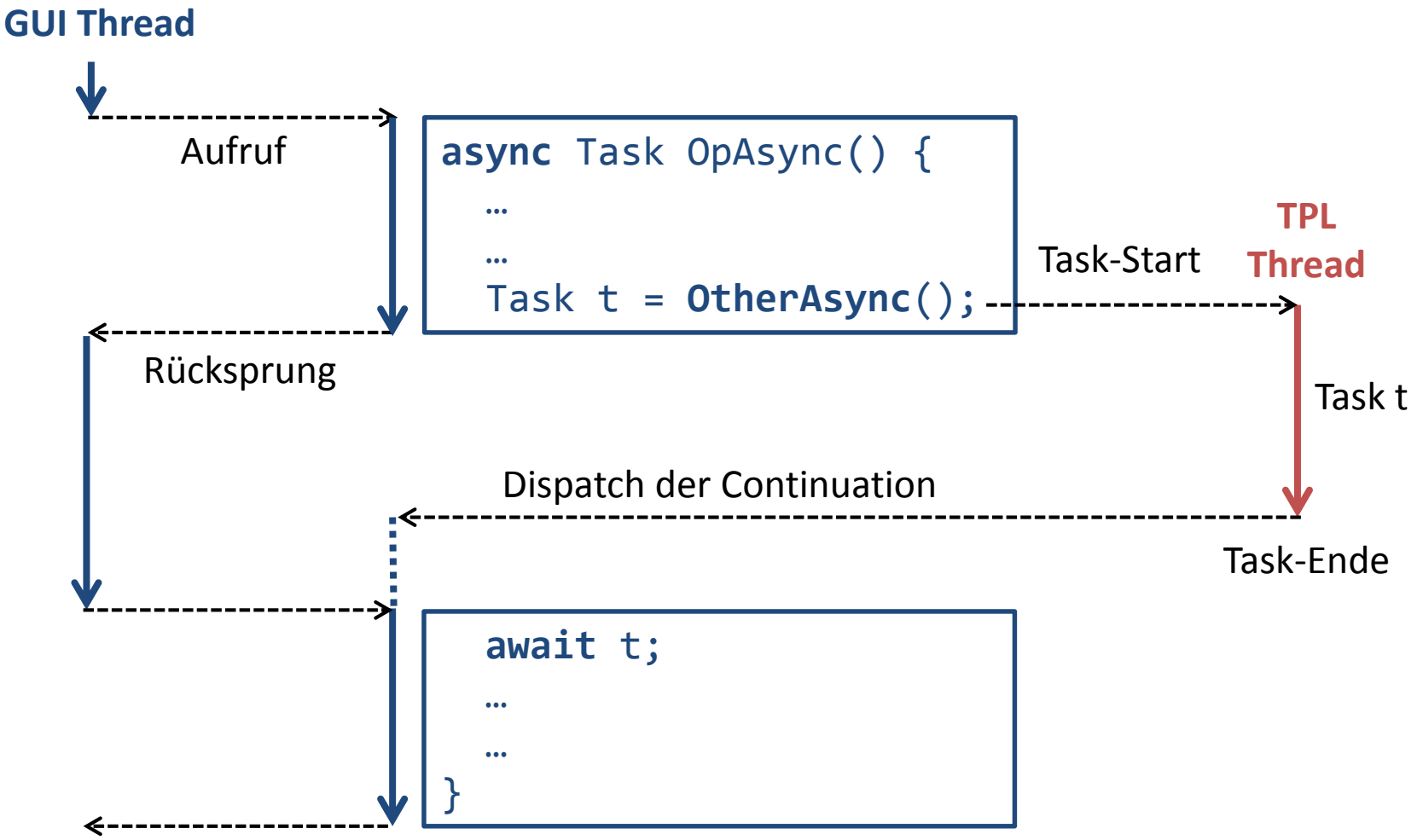
Fall 1: Kein Synchronisationskontext

- Task-Thread führt Abschnitt nach Await aus



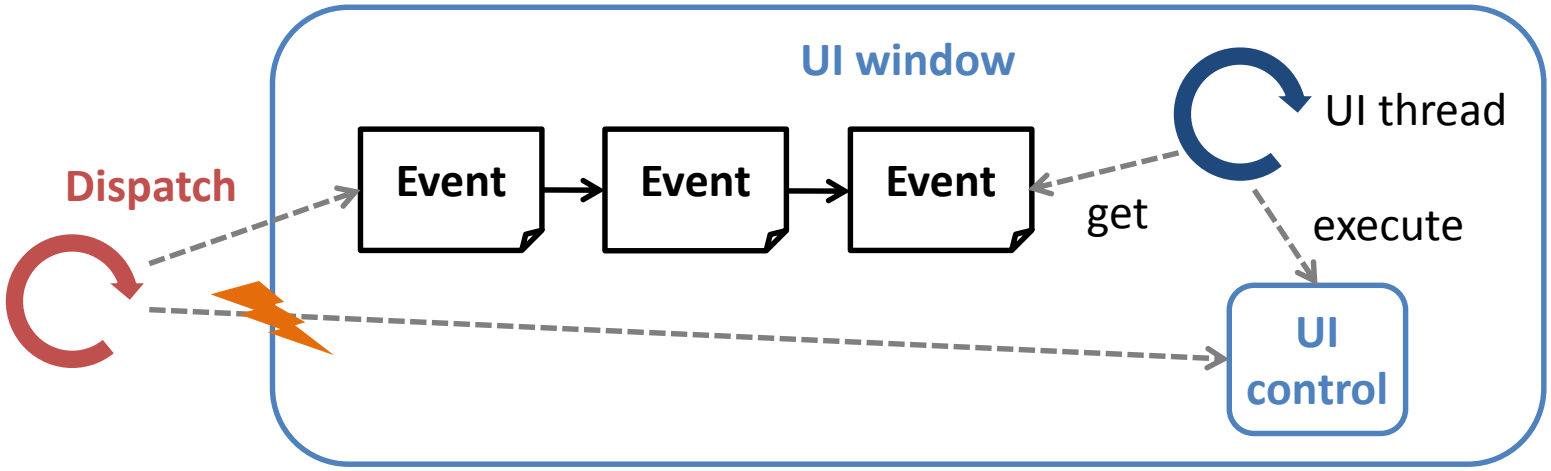
Fall 2: Mit Synchronisationskontext

- Beispiel GUI-Thread als Aufrufer: Dispatch



Wieso diese Komplexität?

- GUI Frameworks sind Single-Threaded
  - Nur dedizierter UI Thread darf UI-Controls zugreifen
  - Threads können Arbeiten als UI-Events beim UI einreihen
  - UI Thread verarbeitet sukzessive diese UI-Events

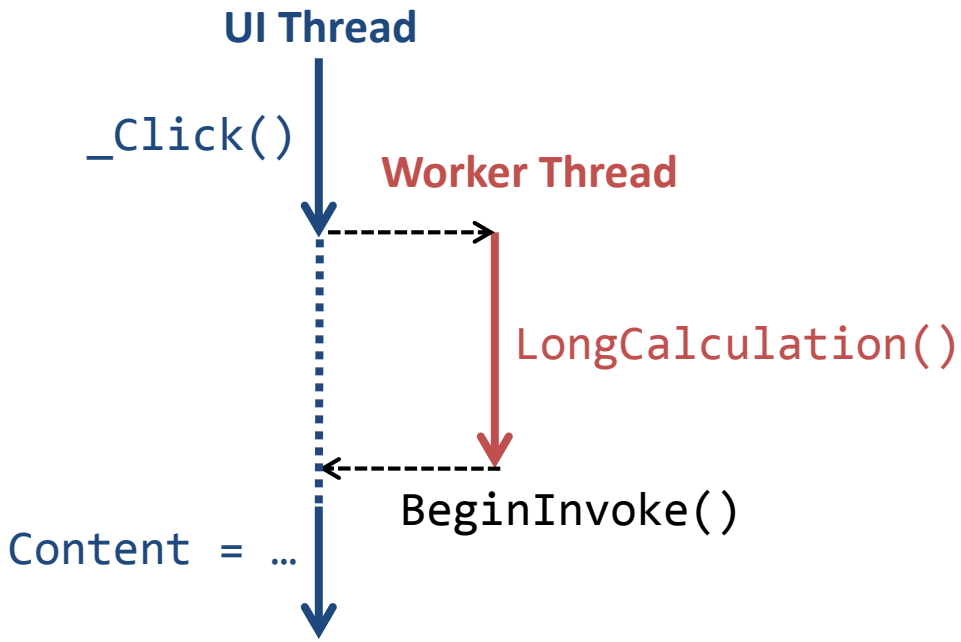


Klassische UI-Programmierung

```
void startCalculationButton_Click(…) {
   new Thread(LongCalculation).Start();
}

void LongCalculation() {
 …calculation…
 Dispatcher.BeginInvoke(new Action(() => {
   resultLabel.Content = …;
 }));
}
```

Langlaufende Operation an Thread outsourcen

Fremder Thread darf UI nicht zugreifen => Dispatch zu UI



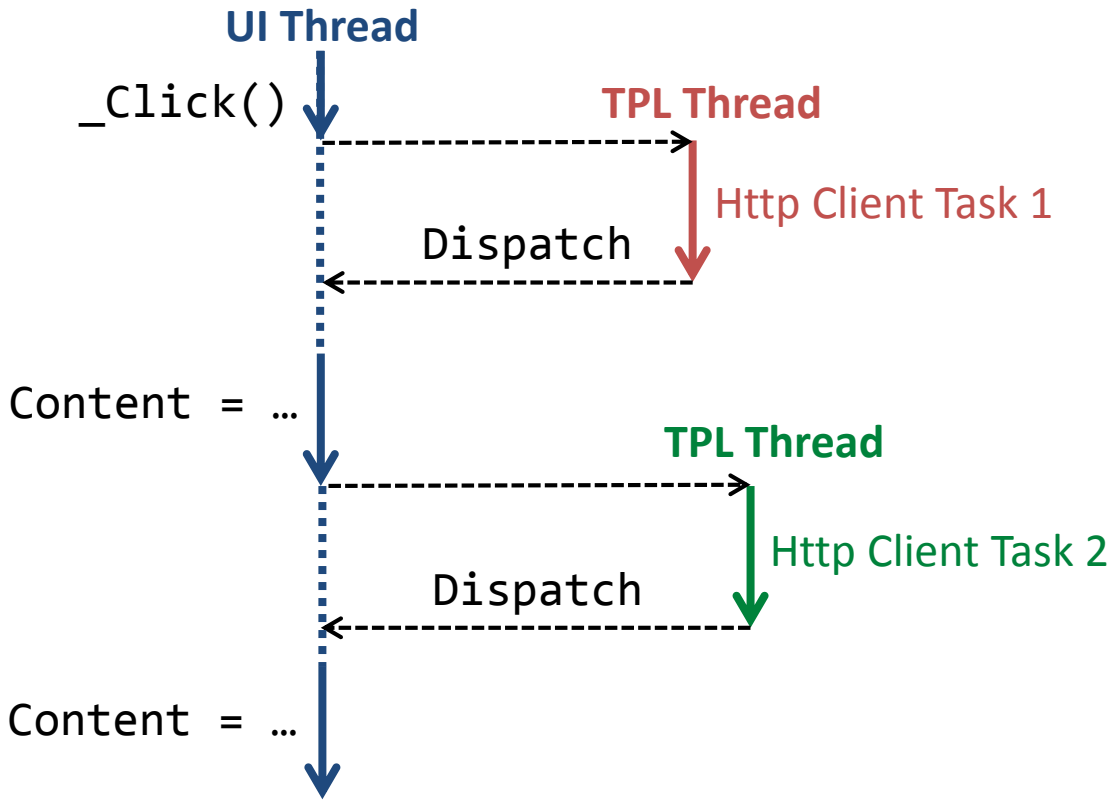
Nicht-Blockierende UIs

- Klassisch: Zerstückelung der Logik
  - Kette von Dispatch (UI Thread/fremder Thread)
  - Hilfsklasse BackgroundWorker
- Leserlicher Code mit Async/Await
  - Logik in einem Guss (eine Methode)
  - Zerstückelung in mehrere UI-Events hinter Kulissen

Async/Await Sequenz

```
async void startDownload_Click(…) {
   HttpClient client = new HttpClient();
   foreach (var url in collection) {
      var data = await client.GetStringAsync(url);
      textArea.Content += data;
   }
}
```

API bietet diverse async Methoden



Tipps zu Async/Await

- async Methode in synchroner Methode aufrufen
  - `task.Wait()` oder `task.Result`
  - ```
    void Test() {
      Task myTask = LongOpAsync();
      myTask.Wait();
    }
    ```
- async Lambda
  - ```
    async () => {
      await SomeOpAsync();
      …
    }
    ```

Achtung Fallstricke

1. Async Methoden sind nicht per se asynchron
2. Thread-Wechsel innerhalb Methode
3. Quasi-Parallelität der UI-Event-Handler
4. Race Conditions bleiben möglich
5. UI-Deadlocks wegen Async Task-Zugriff

Weitere…

Pitfall 1

- Async Methoden sind nicht per se asynchron

```
public async Task<bool> IsPrimeAsync(long number) {
 for (long i = 2; i <= Math.Sqrt(number); i++) {
  if (number % i == 0) { return false; }
 }
 return true;
}
```

Läuft vollständig synchron

Aufrufer ist während gesamter Ausführung blockiert

Compiler-Warnung: kein Await in Async Methode

Pitfall 1: Workaround

- Rechenintensive Operation explizit als Task ausführen

```
public async Task<bool> IsPrimeAsync(long number) {
  return await Task.Run(() => {
     for (long i = 2; i <= Math.Sqrt(number); i++) {
       if (number % i == 0) { return false; }
     }
     return true;
  });
}
```

Pitfall 1: Diskussion

- Muss situativ explizit Task in async lancieren
  - Um Asynchronität zu erhalten
- Compiler-Warnung ist nicht hinreichend

```
public async Task ComputeAsync() {
  var result = VeryLongCalculation();
  await UploadResultAsync(result);
}
```

Läuft immer noch auf Kosten des Aufrufers

Keine Compiler-Warnung

Pitfall 2

- Thread-Wechsel innerhalb Methode
  - Falls kein UI / Synchronisationkontext mit Dispatcher

```
public async Task DownloadAsync() {
  Console.WriteLine("BEFORE "+ Thread.CurrentThread.ManagedThreadId);
  HttpClient client = new HttpClient();
  Task<string> task = client.GetStringAsync("...");
  string result = await task;
  Console.WriteLine("AFTER " + Thread.CurrentThread.ManagedThreadId);
}
```

Partielle Nebenläufigkeit berücksichtigen
Achtung bei Thread-lokalen Variablen

```
BEFORE 10
…
AFTER 14
```

Pitfall 3

- Quasi-Parallelität der UI-Event-Handler
  - Im UI: Bei jedem Await können natürlich andere UI-Events dazwischen laufen

```
async void startDownload_Click(…) {
   HttpClient client = new HttpClient();
   foreach (var url in collection) {
      var data = await client.GetStringAsync(url);
      textArea.Content += data;
   }
}
```

User-Click dazwischen => collection evtl. verändert

Muss Interferenzen unterbinden
- **Snapshots / Isolation von anderen Events**
- **Zwischenevent-Ausführung einschränken**

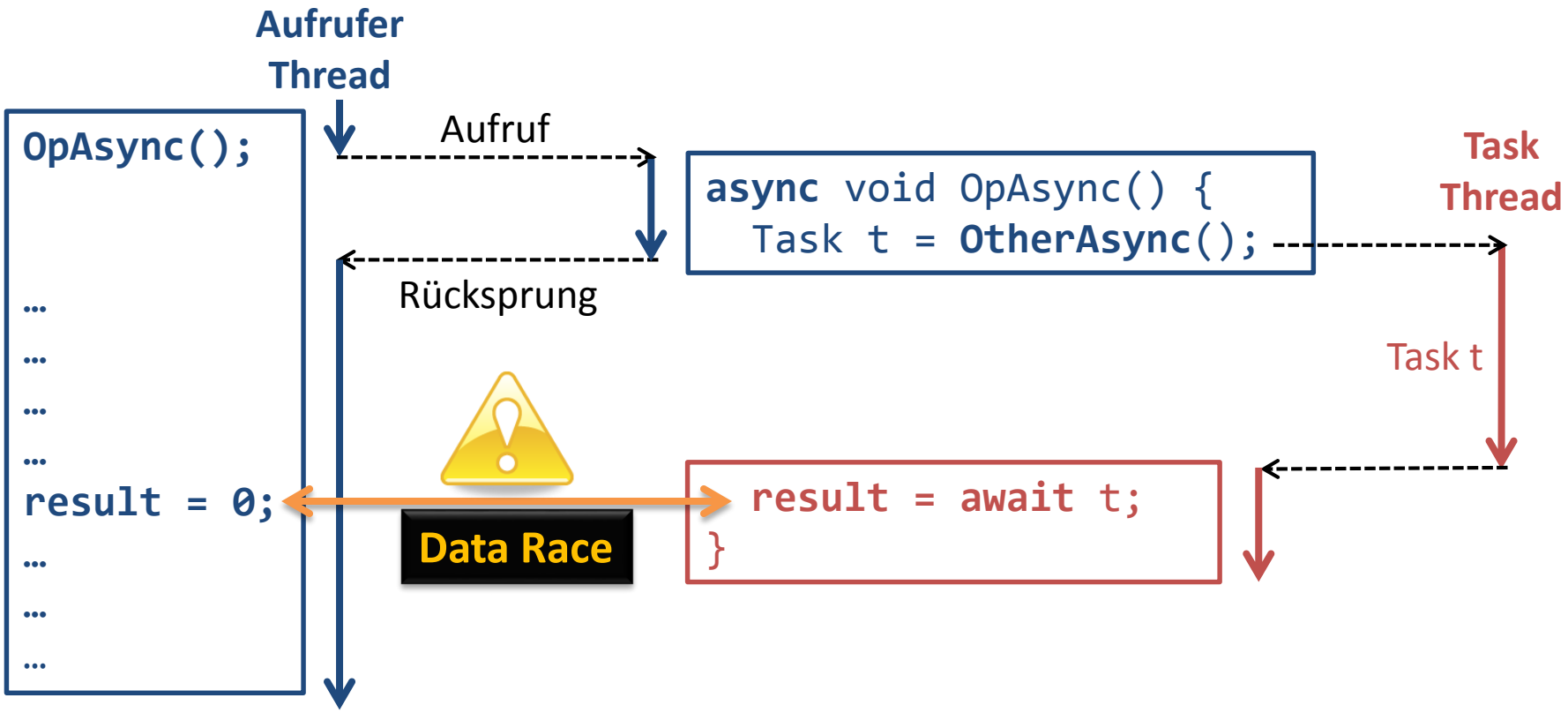
Pitfall 4

"The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better [...] because the code is simpler and you don't have to guard against race conditions"

- http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx

- Race Conditions bleiben natürlich möglich!
  - Abschnitt nach await läuft potentiell parallel zum Aufrufer
    - Falls ohne Synchronisationkontext (z.B. kein UI-Thread)
    - Falls `configureAwait(false)`
  - Explizite gestartete Task (`Task.Run()`, Threads etc.)

Pitfall 4: Race Condition



Pitfall 5

- UI-Deadlocks
  - Aufruf von Task.Wait() oder Task.Result in UI-Thread

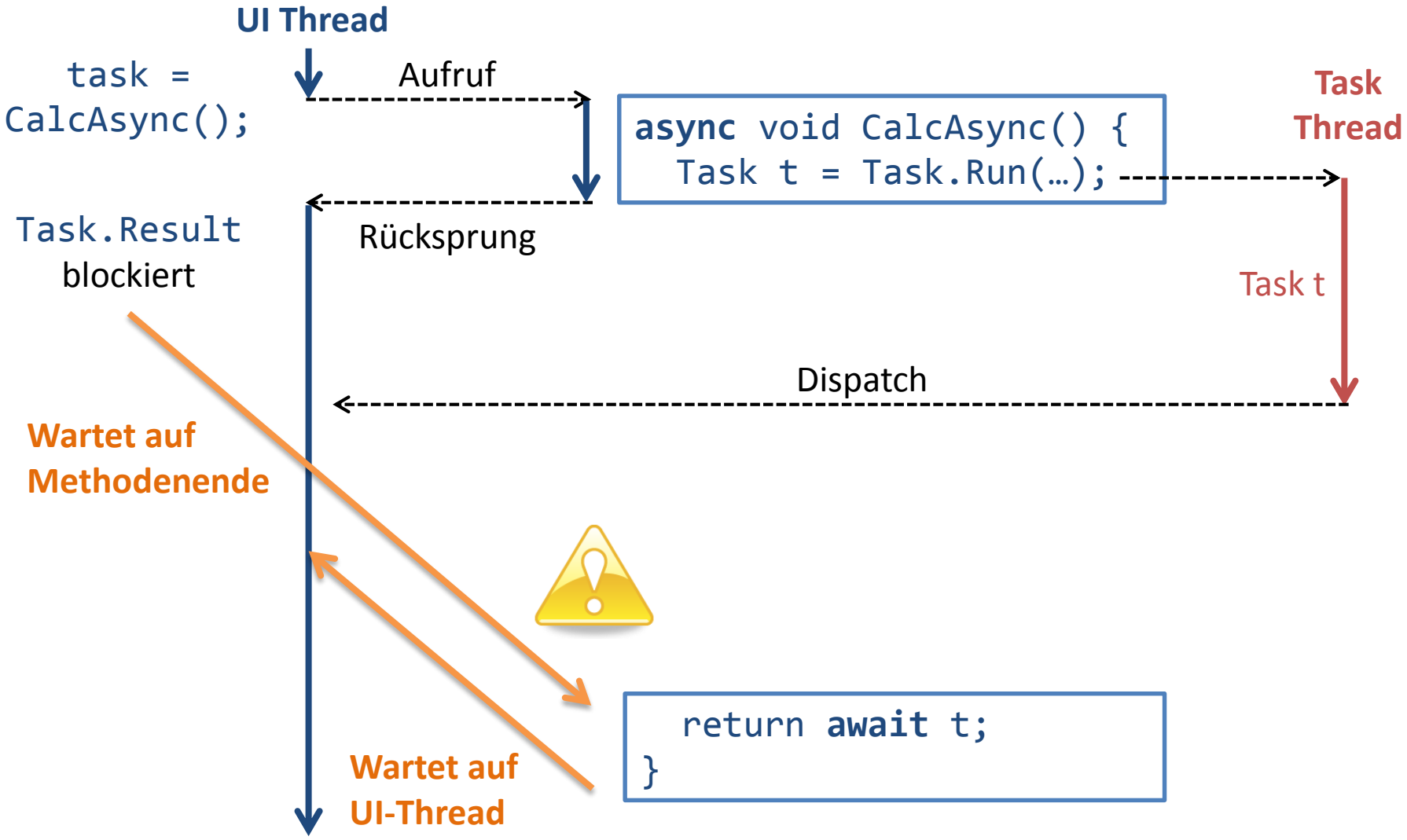
```
void calculationButton_Click(…) {
    Task<string> task = CalculateAsync();
    label.Content = task.Result;
}
```

Implizites Task.Wait()
=> Blockiert UI Thread

```
async Task<string> CalculateAsync() {
    return await Task.Run(…);
}
```

Abschnitt nach await kann nicht laufen, weil UI-Thread blockiert ist

Pitfall 5: Analyse



Weitere Pitfalls

- Für «Fire-and-forget» Tasks und async Methoden
  - Exceptions in diesen Tasks werden ignoriert
    - Ab .NET 4.5
  - Tasks laufen nicht garantiert zu Ende
    - Weil TPL Worker Threads Background Threads sind

Schlussfolgerungen

- Komplizierter Mechanismus
  - Anders als gewöhnliche asynchrone Programmierung
  - Genaues Verständnis unabdingbar
  - Diverse Fallstricke
- Wohlüberlegter Einsatz
  - Primär für UI-Layer angemessen
  - Sonst schnell hohe Komplexität

Danke für Ihr Interesse

- .NET Concurrency Industriekurse
  - http://concurrency.ch/Training
- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    IFS Institut für Software
    Rapperswil, Schweiz
  - lblaeser@hsr.ch
  - http://ifs.hsr.ch
    http://concurrency.ch