

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: Hinweis

- 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**

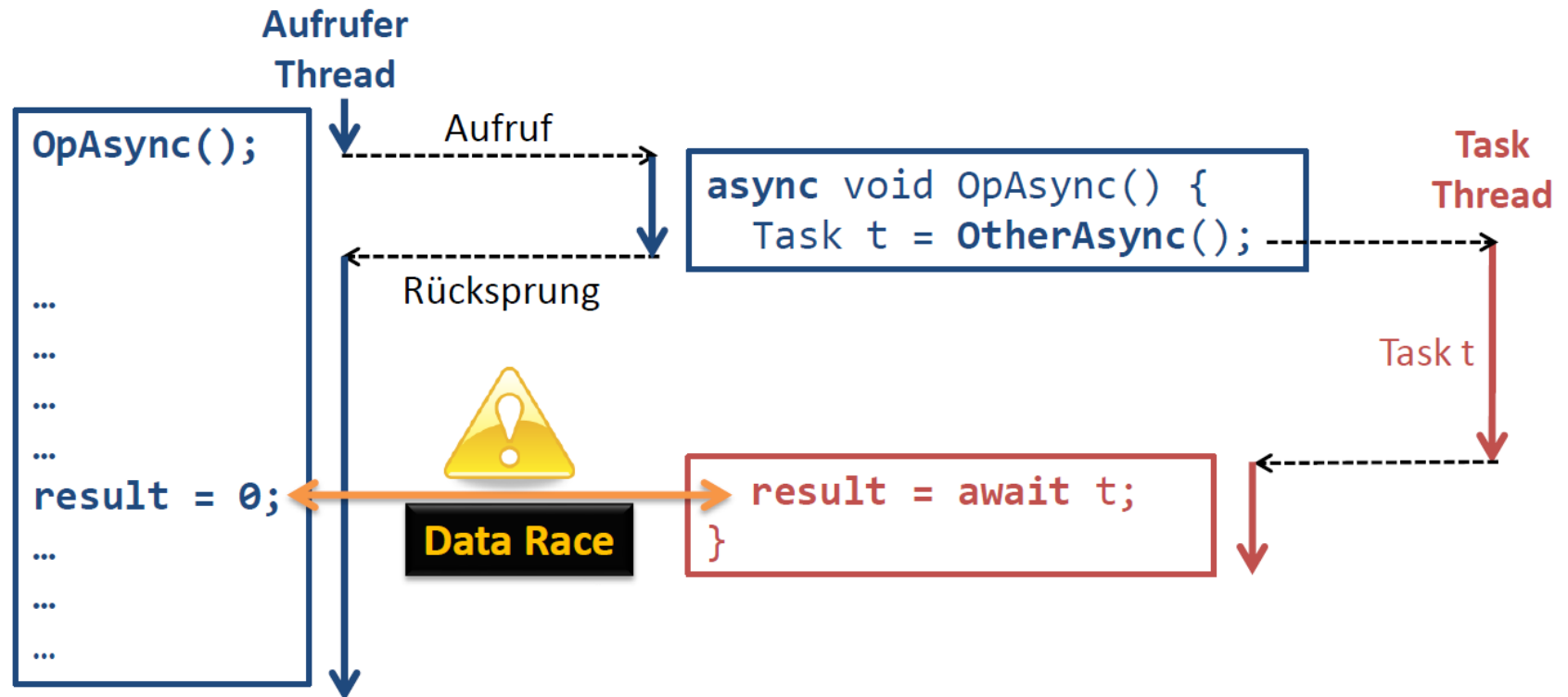
Pitfall 4

“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

