

Praktische Informatik

Vorlesung 12

Nebenläufigkeit

Zuletzt haben wir gelernt...

- Wie die Plattform Xamarin funktioniert.
- Wie man mit Xamarin.Forms plattformunabhängige Apps entwickeln kann.
- Die Unterschiede zu WPF anhand des Beispiels Temperaturumrechner.
- Was die Klasse ContentPage, Layouts und die Klasse OnPlatform leisten.
- Wie man in Xamarin.Forms mit Element Binding und Ressourcen umgeht.
- Weitere Unterschiede zur WPF anhand des Beispiels Mitgliederverwaltung.

Inhalt heute

- Multi Threading in C#
- Abhängigkeiten zwischen Threads
- Synchronisierung
- Monitor und Semaphore
- Erzeuger-Verbraucher-Problem
- Threads und WPF
- Die Task Parallel Library
- Die Klasse Task
- Asynchrone Programmierung

Langlaufende Aufgaben

- In Anwendung kommt es häufiger vor, dass bestimmte Dinge viel Zeit in Anspruch nehmen.
 - z.B. Versenden einer E-Mail, längere Berechnungen, ...
- Werden solche Operationen in unserer Anwendung synchron (nacheinander) behandelt, friert die Benutzeroberfläche ein.
 - Es können keine Eingabe mehr entgegengenommen werden.
- Dies ist natürlich wenig benutzerfreundlich.
 - Es wäre besser, die Aufgabe könnten quasi im Hintergrund bearbeitet werden.

Multi Threading

- In der Veranstaltung „Betriebssysteme“ haben wir gelernt, dass jeder Anwendung ein **Prozess** zugeordnet ist.
 - Ein Prozess führt wiederum mindestens einen **Thread** aus.
- Innerhalb eines Prozesses können auch mehrere Threads ausgeführt werden.
 - Dann sorgt der Scheduler dafür, dass die CPU-Zeit in schneller Abfolge (z.B. alle 20 ms) auf die Threads verteilt wird.
- In **multi-Threading** Anwendung kann die Bearbeitung von mehreren Operationen daher quasi **gleichzeitig (engl. concurrent)** erfolgen.

Multi-Threading in C#

- Nehmen wir an, wir haben eine Methode, die eine lange Berechnung durchführt:

```
public static void BackgroundWorker()  
{  
    while (true)  
    {  
        Console.WriteLine("Hello, World!");  
        Thread.Sleep(500);  
    }  
}
```

Thread.Sleep hält den aktuellen Thread für n Millisekunden an.

- Diese Methode können wir nun in einem eigenen Thread ausführen.
 - Wir erzeugen dazu ein neues Objekt der Klasse Thread.
 - Die Methode Start lässt den neuen Thread laufen.

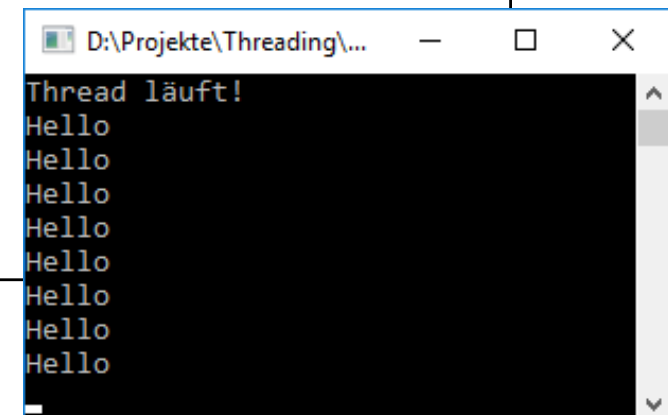
```
Thread t = new Thread(BackgroundWorker);  
t.Start();
```

Anonyme Methode

- Dem Konstruktor der Thread-Klasse kann auch eine anonyme Methode übergeben werden.
 - Dies macht die Erzeugung eines Threads noch einfacher:

```
var t1 = new Thread(() =>
{
    while (true)
    {
        Console.WriteLine("Hello");
        Thread.Sleep(500);
    }
});

t1.Start();
Console.WriteLine("Thread läuft!");
```



```
D:\Projekte\Threading\...
Thread läuft!
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

Mehrere Threads

- Der Aufruf der Thread-Methode `Start()` kommt sofort wieder zurück.
 - Die Aufgabe innerhalb der Methode `BackgroundWorker` bzw. der anonymen Methode wird im Hintergrund ausgeführt.
 - Solange ein solcher Thread arbeitet, wird die Anwendung nicht beendet.
- Die Anwendung kann auch mehrere Thread-Objekte erzeugen.
 - Alle Aufgaben werden dann quasi gleichzeitig abgearbeitet.
 - Der Thread-Scheduler verteilt die CPU-Zeit auf die Threads.
- **Achtung:**
 - Wenn mehrere CPU-Cores vorhanden sind, werden Threads nicht notwendigerweise gleichmäßig verteilt!
 - Gleichzeitigkeit (concurrency) bedeutet nicht automatisch Parallelität!

Zugriff auf Daten

- Threads haben Zugriff auf alle Daten des aktuellen Prozesses.
 - Entsprechend kann man auch Variablen innerhalb eines Threads verändern.
- Das folgende Beispiel zählt einen Zähler innerhalb eines Threads von 0 bis 10:

```
private static int counter = 0;

static void Main()
{
    var t1 = new Thread(CountUp);
    t1.Start();
    Console.WriteLine("Zähle hoch!");

    Console.ReadLine();
}
```

```
public static void CountUp()
{
    while (counter < 10)
    {
        counter++;
        Console.WriteLine(counter);
        Thread.Sleep(10);
    }
}
```

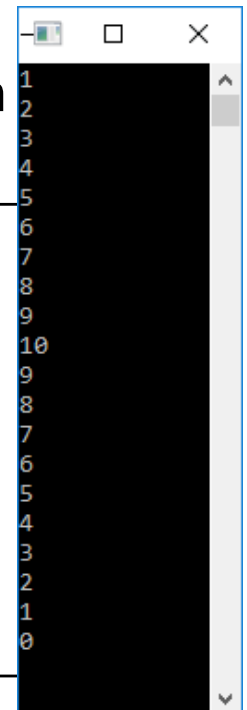
Abhängigkeiten

- Wenn mehrere Threads auf den selben Daten operieren, können Abhängigkeiten entstehen.
 - Ein Thread muss dann auf einen anderen warten.
- Um auf das Bearbeitungsende eines Threads zu warten, existiert die Methode `Join()`.
 - Wir können z.B. auf den Hochzähl-Thread warten, um danach einen anderen Thread zu starten:

```
static void Main()
{
    var t1 = new Thread(CountUp);
    var t2 = new Thread(CountDown);

    t1.Start();
    t1.Join();
    t2.Start();

    Console.ReadLine();
}
```



Thread-Prioritäten

- Normalerweise sind alle Threads gleichberechtigt.
 - Alle Threads erhalten dann vom Scheduler gleich große Zeitscheiben.
- Ein Thread-Objekt besitzt allerdings die Eigenschaft `Priority`.
 - In den Stufen Highest bis Lowest kann darüber die Priorität des Threads verändert werden.
- Achtung: Die Thread-Priorität sollte nur in gut begründeten Fällen verändert werden.

Buchstaben ausgeben

- Im folgenden Beispiel geben zwei Threads den Buchstaben ‚A‘ und ‚B‘ auf der Konsole aus.
 - Ohne Veränderung der Prioritäten haben beide Thread gleich viel Zeit, ihre Buchstaben wechselseitig auszugeben.
- Nun wird die Priorität des ersten Threads gegenüber des zweiten erhöht.
 - Es lässt sich beobachten, dass ein höher priorisierter Thread häufiger in der Lage ist, seinen Buchstaben auszugeben:

```
var t1 = new Thread(() => { for (int i = 0; i < 5000; i++) { Console.Write("A"); } });  
var t2 = new Thread(() => { for (int i = 0; i < 5000; i++) { Console.Write("B"); } });  
  
t1.Priority = ThreadPriority.Highest;  
t2.Priority = ThreadPriority.Lowest;  
  
t1.Start();  
t2.Start();  
  
Console.ReadLine();
```

Probleme mit Multi-Threading

- Arbeiten mehrere Threads auf den selben Ressourcen, kann es zu einer ganz neuen Kategorie von Problemen kommen.
 - Solche Probleme sind oft sehr schwierig zu erkennen und zu reproduzieren.
- Ein Problem kann entstehen, wenn Threads mehrere Ressourcen gleichzeitig benötigen.
 - Hierbei kann es zu einer **Verklemmung (engl. Deadlock)** kommen, wenn Threads Ressourcen sperren.
 - Andere Threads warten dann (unendlich lang), bis alle benötigten Ressourcen zur Verfügung stehen.
- Ein weiteres Problem kann entstehen, wenn die Ausführung eines Threads an einer **ungünstigen Stelle unterbrochen** wird.
 - Die Unterbrechung sorgt dann evtl. für ungültige Daten, mit dem ein anderer Thread dann weiter arbeitet.

Klasse Bankkonto

- Denken wir zurück an unsere Klasse Bankkonto:

```
class Bankkonto
{
    private double kontostand = 0;

    public void Einzahlen(double betrag)
    {
        kontostand += betrag;
    }

    public void Auszahlen(double betrag)
    {
        if (kontostand >= betrag)
        {
            kontostand -= betrag;
        }

        if (kontostand < 0)
            throw new Exception("Betrag kleiner 0!");
    }
}
```

Auszahlungen sind nur möglich, wenn genug Geld auf dem Konto vorhanden ist.

Zur Sicherheit werfen wir eine Ausnahme, sollte das Unmögliche einmal eintreten.

Ein- und Auszahler

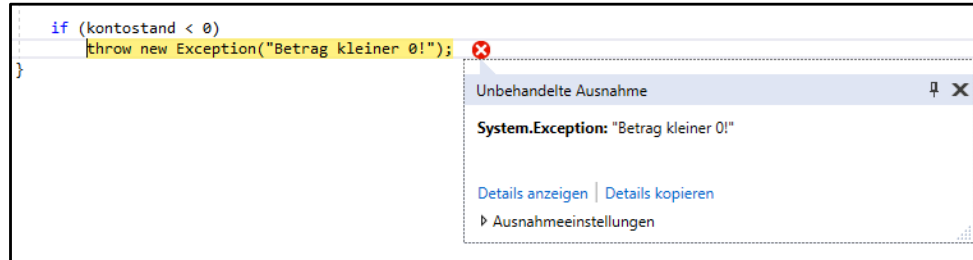
- Ein Bankkonto-Objekt soll nun durch mehrere Threads benutzt werden.
 - Dazu erstellen wir zwei Methoden Einzahler und Auszahler, die jeweils 1000€ ein- bzw. auszahlen.
 - Jede Methode wartet danach eine zufällige Anzahl von Millisekunden.

```
private void Einzahler()
{
    while (true)
    {
        konto.Einzahlen(1000);
        Thread.Sleep(rnd.Next(0, 500));
    }
}
```

```
private void Auszahler()
{
    while (true)
    {
        konto.Ausszahlen(1000);
        Thread.Sleep(rnd.Next(0, 500));
    }
}
```

Ausnahme

- Wir lassen nun 50 Einzahler- und 200 Auszahler-Threads laufen.
 - Nach kurzer Zeit wird eine Ausnahme geworfen.



- Der eigentlich unmögliche Fall ist eingetreten.
 - Das Bankkonto wurde überzogen!

Kritischer Abschnitt

- Wir sehen uns den Programmcode der Auszahlen-Methode in der Bankkonto-Klasse genauer an.

1	<code>if (kontostand >= betrag)</code>
2	<code>kontostand -= betrag;</code>

- Folgendes Problem kann hier entstehen:
 - Thread 1 läuft in Zeile 1 und wird in Zeile 2 vorgelassen, da der Betrag auf dem Konto ausreicht.
 - In diesem Moment kommt auch Thread 2 an diesem Punkt an und wird ebenfalls in den Abschnitt vorgelassen.
 - Beide Threads zahlen jeweils 1000€ aus und überziehen damit das Konto, obwohl dies nicht erlaubt ist.

Synchronisation

- Im letzten Beispiel haben wir gesehen, dass es gefährlich sein kann, wenn bestimmte Abschnitte eines Programms durch mehr als einen Thread durchlaufen werden.
 - Ein solcher Abschnitt wird als **kritischer Abschnitt** bezeichnet.
- Diese Abschnitte müssen also geschützt werden.
 - Eine einfache Möglichkeit ist der **wechselseitige Ausschluss** (*engl. mutual exclusion*).
 - Es kann dann jeweils nur ein Thread diesen Programmabschnitt durchlaufen.
- Ein solcher wechselseitiger Ausschluss kann mit der Anweisung `lock`, bzw. mit der Klasse `Monitor` erreicht werden.

Monitor und lock

- Wir wollen den kritischen Bereich unseres Bankkontos nun schützen, so dass ihn nur jeweils ein Thread betreten kann.
 - Die Verwendung von Monitor und lock ist äquivalent:

```
Monitor.Enter(this);

if (kontostand >= betrag)
{
    kontostand -= betrag;
}

Monitor.Exit(this);
```



```
lock (this)
{
    if (kontostand >= betrag)
    {
        kontostand -= betrag;
    }
}
```

- Durch diese Änderung kann der Fehler mit einem überzogenen Konto nicht mehr eintreten.

Semaphore

- Ein Semaphore beschränkt den Zugriff auf einen kritischen Bereich nicht nur auf einen Thread.
 - Die Anzahl der zulässigen Threads ist konfigurierbar.
- Das folgende Beispiel sorgt dafür, dass der kritische Bereich nur von maximal 3 Threads gleichzeitig betreten werden kann.

```
class ClassWithSemaphore
{
    private static Semaphore semaphore = new Semaphore(3, 3);
    private int counter = 0;

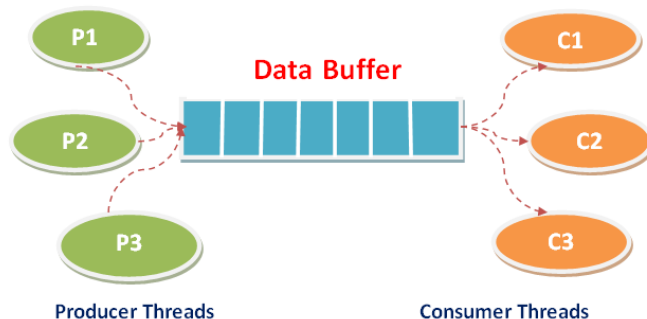
    public void DoSomething()
    {
        semaphore.WaitOne();
        counter++;
        Console.WriteLine(counter);
        Thread.Sleep(50);
        counter--;
        semaphore.Release();
    }
}
```

WaitOne wartet so lange, bis weniger als 3 Threads im kritischen Bereich sind.

Release teilt mit, dass wieder ein Thread eintreten kann.

Erzeuger-Verbraucher

- Semaphore können helfen dabei, das sog. **Erzeuger-Verbraucher-Problem** zu lösen.
 - Viele Erzeuger legen Daten in einem begrenzt großen Speicher ab.
 - Gleichzeitig entnehmen viele Verbraucher Elemente aus diesem Speicher.



- Es muss folgendes sichergestellt werden:
 - Erzeuger müssen warten, wenn der Speicher voll ist.
 - Verbraucher müssen warten, wenn der Speicher leer ist.

Klasse Queue

- Wir erstellen eine Klasse Queue, als Datenspeicher, die von Erzeugern und Verbrauchern benutzt werden soll.
 - Die zu speichernden Werte legen wir in einem Array ab.
 - Die Größe des Arrays übergeben wir im Konstruktor.
- Die Methode Push soll einen Wert ablegen.
 - Dabei kann nur ein weiterer Wert abgelegt werden, wenn noch Platz vorhanden ist.
 - Dazu nutzen wir eine Semaphore free.
- Die Methode Pop soll einen Wert entnehmen.
 - Es kann nur ein Wert entnommen werden, wenn noch Werte vorhanden sind.
 - Dazu nutzen wir eine Semaphore used.

Klasse Stack

```
class Stack
{
    private int[] memory;
    private int pos = -1;
    private Semaphore free;
    private Semaphore taken;

    public Stack(int size)
    {
        memory = new int[size];
        free = new Semaphore(size, size);
        taken = new Semaphore(0, size);
    }

    public int Count
    {
        get { return pos+1; }
    }
}
```

```
public void Push(int value)
{
    free.WaitOne();
    pos++;
    memory[pos] = value;
    taken.Release();
}

public int Pop()
{
    taken.WaitOne();
    var value = memory[pos];
    pos--;
    free.Release();
    return value;
}
```

Erzeugen und verbrauchen

- Wir können nun beliebig viele Erzeuger und Verbraucher in Threads auf die Queue zugreifen lassen.
- Diese legen Daten in der Queue ab oder entnehmen dort Daten.
- Dank der Semaphore werden die Threads angehalten, wenn nötig.
- Die Speichergrenzen der Queue werden nie verletzt.

```
static void Main()
{
    for (int i = 0; i < 100; i++)
        new Thread(Producer).Start();

    for (int i = 0; i < 10; i++)
        new Thread(Consumer).Start();

    Console.ReadLine();
}

private static void Producer()
{
    while (true)
    {
        q.Push(rnd.Next(0, 1000));
        Thread.Sleep(rnd.Next(0, 500));
    }
}

private static void Consumer()
{
    while (true)
    {
        q.Pop();
        Thread.Sleep(rnd.Next(0, 500));
    }
}
```


Threads und WPF

- Auch WPF-Anwendungen können Threads benutzt werden, um Hintergrundaktivitäten zu bearbeiten.
 - WPF-Anwendungen bestehen bereits aus mehreren Threads, der wichtigste ist der sog. UI-Thread.
- Soll aus einem anderen Thread als dem UI-Thread auf Elemente der Oberfläche zugegriffen werden, muss der sog. **Dispatcher** benutzt werden.
 - Der Dispatcher hat Zugriff auf die Nachrichtenschleife der Oberfläche und kann mit den Elementen dort kommunizieren.
- Alle Interaktionselemente besitzen die Eigenschaft **Dispatcher**.
 - Den Methoden `Invoke` und `BeginInvoke` kann ein `Delegate` übergeben werden, der die gewünschte Aktion im UI-Thread ausführt.

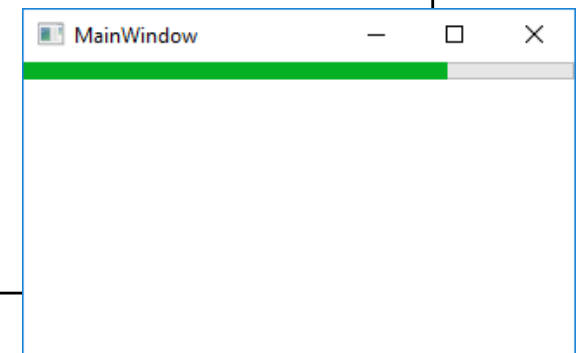
ProgressBar

- In einer WPF-Oberfläche ist ein **Fortschrittsbalken (engl. progress bar)** abgelegt.

```
<StackPanel>
    <ProgressBar x:Name="progress" Minimum="0" Maximum="100" Height="10" Value="0"/>
</StackPanel>
```

- In einem Thread wollen wir nun den Fortschritt erhöhen.
 - Dazu nutzen wir den Dispatcher des Fensters, um die Eigenschaft Value des Fortschrittsbalkens zu verändern.

```
new Thread(() =>
{
    for (int i = 0; i < 100; i++)
    {
        Dispatcher.Invoke(() => { progress.Value++; });
        Thread.Sleep(20);
    }
}).Start();
```



Nachteile der Thread-Klasse

- Threads ermöglichen es, Aufgaben gleichzeitig in einer Anwendung zu bearbeiten.
 - Die Thread-Klasse erlaubt dabei große Kontrolle über solche Aufgaben.
 - Allerdings hat die Verwendung der Klasse einige Nachteile.
- Threads sind bei der **Erzeugung** sehr **ressourcenintensiv**.
 - Daher existiert auch der sog. Thread-Pool, eine Menge von vordefinierten Threads, die für Hintergrundaktivitäten wieder verwendet werden können.
 - Leider ist der Umgang mit dem Thread-Pool wenig intuitiv.
- Das **Umschalten** zwischen den Threads **kostet** CPU-Zeit.
 - Bei Single-Core Prozessoren kann das dazu führen, dass sich die gesamte Bearbeitungszeit einer Aufgabe mit Threads sogar erhöht.

Task Parallel Library

- Die Verwendung von Threads kann die Performance von Anwendungen sogar verschlechtern.
 - Um den Umgang mit Parallelität zu verbessern, wurden mit .Net 4.0 zusätzliche Möglichkeiten nachgerüstet.
 - **Die Task Parallel Library (TPL).**
- Die wichtigste neue Klasse ist **Task**.
 - Diese Klasse repräsentiert eine Aufgabe.
 - Sie ist quasi ein Versprechen (engl. promise) der späteren Ausführung.
- Bei einem Task wird dynamisch entschieden, ob für diese Aufgabe ein Thread benutzt werden soll, oder nicht.
 - Um Ressourcen zu schonen, wird unter der Haube zudem der Thread-Pool benutzt.

Task

- Ein Task kann sehr einfach erzeugt und gestartet werden.

```
Task t1 = new Task(DoSomeWork);  
t1.Start();  
t1.Wait();
```

Die Methode DoSomeWork wird im Hintergrund ausgeführt. Mit t1.Wait() wird auf das Ende der Aufgabe gewartet.

- Auch mehrere Tasks sind kein Problem:

```
Task t1 = Task.Factory.StartNew(DoSomeWork);  
Task t2 = Task.Factory.StartNew(DoSomeWork);  
Task t3 = Task.Factory.StartNew(DoSomeWork);  
Task.WaitAll(t1, t2, t3);
```

Es wird darauf gewartet, bis alle drei Tasks beendet wurden.

- Tasks können auch miteinander verkettet werden:

```
Task.Factory.StartNew(DoSomeWork).ContinueWith(DomSomeMoreWork);
```

DoSomeMoreWork wird erst gestartet, wenn DoSomeWork beendet ist.

Ergebnis eines Tasks

- Eine Task kann auch Ergebnisse zurückgeben.
 - Dazu wird die generische Variante eines Tasks benutzt.
 - Dabei wird der Datentyp des Rückgabewertes angegeben.

```
Task<int> t = Task<int>.Factory.StartNew(() =>
{
    Task.Delay(500);
    return 42;
});

Console.WriteLine(t.Result);
```

- Der Task liefert nach 500 *ms* den Wert 42.
 - Erst danach wird das Ergebnis auf der Konsole ausgegeben

Task als Ergebnis

- Ein Task kann auch selbst das Ergebnis einer Methode sein.
 - Die Methode liefert dann quasi ein Versprechen für die spätere Lieferung eines Ergebnisses.
- Die folgende Methode gibt ein Task-Objekt zurück.
 - Dieser Task liefert nach 500 ms die 42 als Ergebnis.

```
public static Task<int> Return42()  
{  
    return Task.Factory.StartNew(() =>  
    {  
        Task.Delay(500);  
        return 42;  
    }));  
}
```

await und async

- Es kommt häufig vor, dass auf das Ergebnis eines Task gewartet werden muss.
 - Dies blockiert dann die weitere Ausführung der Anwendung.
- Die Anwendung reagiert dann evtl. nicht auf Eingaben.
 - Das ist wenig benutzerfreundlich.
- Mit der TPL wurde der Operator `await` eingeführt.
 - Mit Hilfe von `await` kann das Warten auf Ergebnisse in den Hintergrund verlagert werden.
 - Die Ausführung der aktuellen Methode wird unterbrochen und erst weitergeführt, wenn das Ergebnis eintrifft.
 - Der Rest der Anwendung läuft dabei weiter.
- Der `await` Operator kann nur in Methoden benutzt werden, die mit dem Schlüsselwort `async` markiert sind.
 - Solche Methoden werden als **asynchrone Methoden** bezeichnet.

Beispiel

- Die Methode `GetResult` liefert ein `Task`-Objekt zurück, welches nach 500 ms eine Antwort liefert:

```
public static Task<int> GetResult()
{
    return Task.Factory.StartNew(() =>
    {
        Thread.Sleep(500);
        return 42;
    });
}
```

- Die asynchrone Methode `PrintResult` wartet mit `await` auf das Ergebnis und gibt dieses auf der Konsole aus.

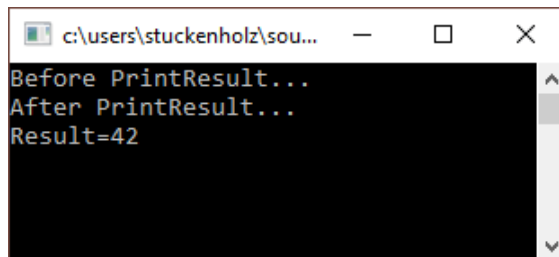
```
public static async void PrintResult()
{
    int result = await GetResult();
    Console.WriteLine("Result=" + result);
}
```

Ausgabe

- Wir benutzen die asynchrone Methode nun wie folgt:

```
Console.WriteLine("Before PrintResult...");  
PrintResult();  
Console.WriteLine("After PrintResult...");
```

- Etwas überraschend ist die Reihenfolge der Ausgaben:



Die asynchrone Methode `PrintResult()` kommt sofort zurück und blockiert die weitere Ausführung der Anwendung nicht. Stattdessen wird der Rest der Anweisungen weiter abgearbeitet. Erst später kommt das Ergebnis von `PrintResult`.

Asynchrone Programmierung

- Asynchrone Methoden helfen bei langlaufenden Aufgaben, nicht blockierende Anwendungen zu schreiben.
 - Dank `async` und `await` sehen diese Methoden kaum anders aus, als ihre synchronen Vertreter.
- Auf Ressourcen, wie Dateien oder das Netzwerk sollten möglichst immer asynchron zugegriffen werden.
 - Das .Net Framework nutzt `async` und `await` an vielen Stellen selbst, z.B. die Klasse `HttpClient`.
- Auch in anderen Programmiersprachen hat die asynchrone Programmierung einen hohen Stellenwert.
 - Der Erfolg von JavaScript mit Node.js ist fast gänzlich darauf zurückzuführen.

Wir haben heute gelernt...

- Wie man Multithreading in C# umsetzt.
- Wie man Abhängigkeiten zwischen Threads organisiert.
- Warum und wie man Threads mit Monitoren und Semaphoren synchronisiert.
- Wie das Erzeuger-Verbraucher-Problem gelöst werden kann.
- Wie man Threads und WPF miteinander in Einklang bringt.
- Wie die Klasse Task aus der Task Parallel Library die Arbeit mit Gleichzeitigkeit vereinfacht.
- Wie man mit Hilfe von async und await Asynchrone Programmierung umsetzt.

Notizen

- Parallel.Foreach