

# Konzepte der Asynchronität in modernen Programmiersprachen

Dr. Peter Dillinger  
[peter.dillinger@email.de](mailto:peter.dillinger@email.de)

---

- Synchrone und asynchrone Methodenaufrufe
- Ergebnisabholung
  - Geteilter Speicher (*Shared Memory*)
  - Future
  - Rückruf (*Callback*)
- Über dem Tellerrand
  - C#
  - JavaScript
  - ABAP
- Zusammenfassung
- Literatur

**Folien, Quellcode, Übungen**  
<https://github.com/phd4S/async>



## Synchroner Programmablauf

- blockierend
- arbeitet sequentiell

## Asynchroner Programmablauf

- nicht-blockierend
- arbeitet parallel
- Erhöht die Reaktionsfähigkeit

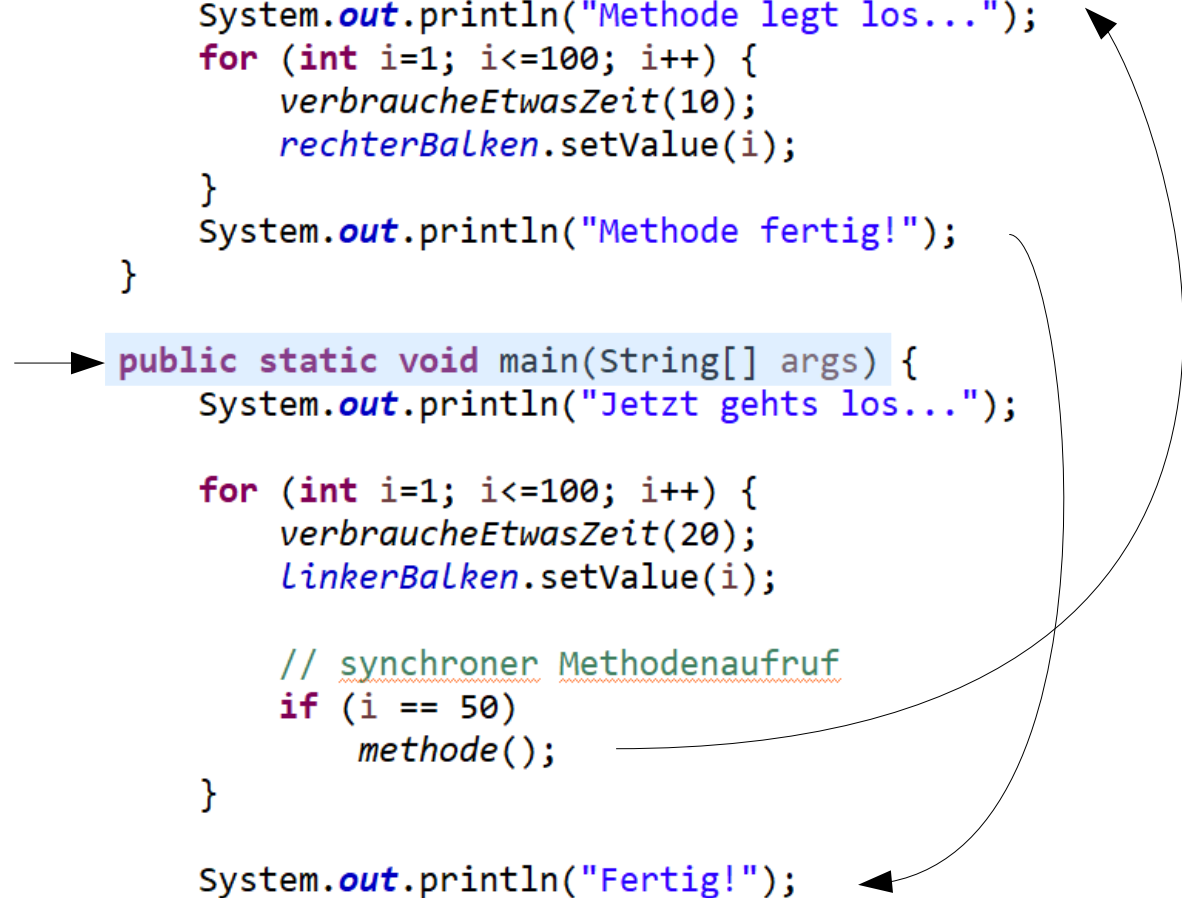
```
// eine normale Methode
public static void methode() {
    System.out.println("Methode legt los...");
    for (int i=1; i<=100; i++) {
        verbraucheEtwasZeit(10);
        rechterBalken.setValue(i);
    }
    System.out.println("Methode fertig!");
}

→ public static void main(String[] args) {
    System.out.println("Jetzt gehts los...");

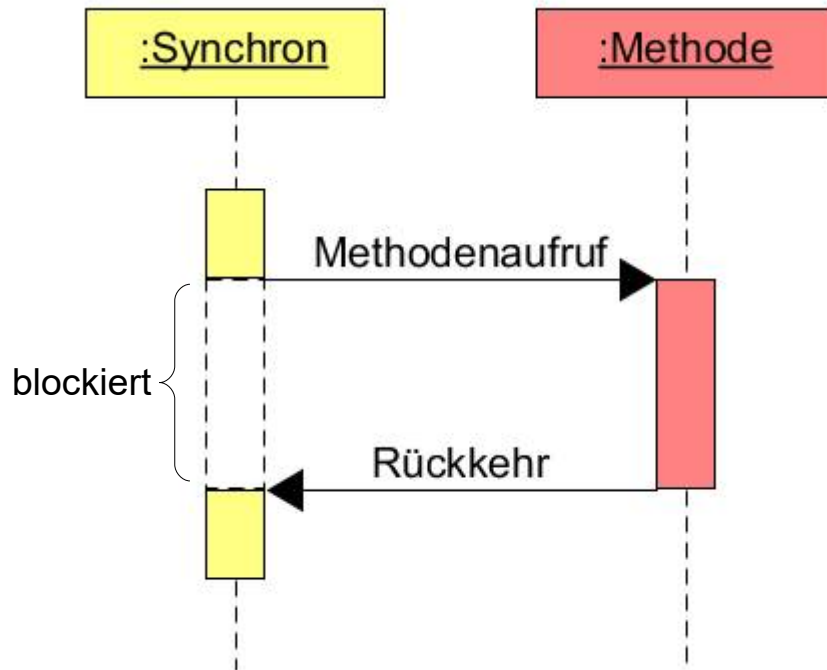
    for (int i=1; i<=100; i++) {
        verbraucheEtwasZeit(20);
        linkerBalken.setValue(i);

        // synchroner Methodenaufruf
        if (i == 50)
            methode();
    }

    System.out.println("Fertig!");
}
```

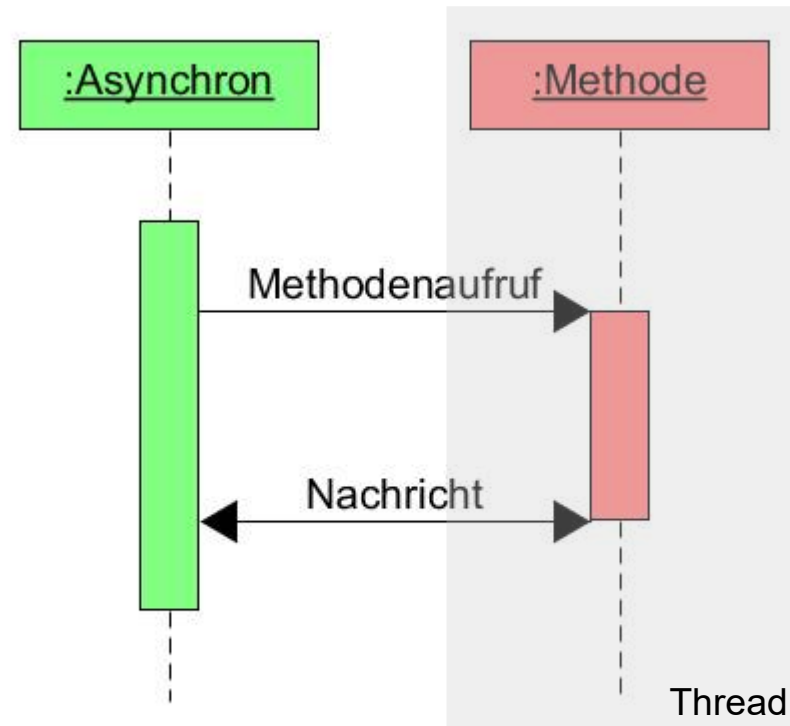


# Methodenaufrufe



## Synchroner Methodenaufruf

- Abgabe der Kontrolle
- nur eine Ausführungseinheit notwendig (Single- oder Multi-Threading)



## Asynchroner Methodenaufruf

- Kontrolle wird beibehalten
- mehrere Ausführungseinheiten notwendig (Multi-Threading)
- Ergebnis muss abgeholt/übermittelt werden (Synchronisation)

- Geteilter Speicher (*Shared Memory*)
  - Ergebnis wird in einer Variable (Attribut) hinterlegt, auf dem beide Methoden Zugriff haben
  - Kann zu Synchronisationsproblemen führen: Wettlaufbedingung (*Race Condition*) und Verlorenes Update (*Lost Update*)
  - Lösung: Sperrung der Variable (Attribut) per Mutex, Semaphoren
  - Kann zu weiteren Synchronisationsproblemen führen: Verklemmung (*Deadlock*) oder Verhungern (*Starvation*)

- Future (*Promises, Delay, Deferred*)  
= Platzhalter (Proxy) für ein Ergebnis
  - noch nicht bekannt bzw.  
Berechnung noch nicht abgeschlossen

```
public interface Future<V> {  
    boolean cancel(boolean mayInterruptIfRunning);  
    boolean isCancelled();  
    boolean isDone();  
    V get() throws InterruptedException, ExecutionException;  
    V get(long timeout, TimeUnit unit)  
        throws InterruptedException, ExecutionException, TimeoutException;  
}
```

- get(...) ist ein blockierender Aufruf
- Führt meist zum Polling:  
    while ( !isDone() ) ...

- Rückruf (Callback, Delegate)
  - nach Abschluss der Methode wird eine andere designierte Methode aufgerufen
  - Aufrufer muss nicht mehr um die Abholung des Ergebnis kümmern, sondern stellt eine gesonderte Methode bereit

```
class CompletableFuture<T> implements Future<T>
```

```
runAsync(...)
```

```
supplyAsync(...)
```

```
thenApply(...)
```

```
thenAccept(...)
```

```
thenRun(...)
```

```
...
```

# Über dem Tellerrand

- C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Task based APM demo");
            Task t = new Task(() =>
            {
                Console.WriteLine("This test is output asynchronously");
            });
            t.Start();
            Console.WriteLine("Task started");

            // Wait task(s) to complete.
            Task.WaitAll(t);
        }
    }
}
```

© <https://www.codeproject.com>



# Über dem Tellerrand

- JavaScript

```
const fetchPromise = fetch('https://mdn.github.io/learning-area/javascript
/apis/fetching-data/can-store/products.json');

console.log(fetchPromise);

fetchPromise.then( response => {
  console.log(`Received response: ${response.status}`);
});

console.log("Started request...");
```

© <https://developer.mozilla.org>

# Über dem Tellerrand

- ABAP

```
"only 10 will be allowed in parallel in the pool  
data(executor) = zcl_executors=>new_fixed_thread_pool( 10 ).
```

```
data(runnable1) = new zcl_my_runnable( datasplit1 ).  
data(runnable2) = new zcl_my_runnable( datasplit2 ).
```

```
"submits individual runnables  
executor->submit( runnable1 ).  
executor->submit( runnable2 ).
```

```
"Asks for a 100 runnables to run but,  
"due to the pool size only 10 will  
"be allowed to be active at a time  
executor->invoke_all( a100Runnables ).
```

```
"blocks executor from queuing new runnables  
executor->shutdown( ).
```

```
"awaits threads to finish and queue to be empty  
executor->await_termination( ).
```

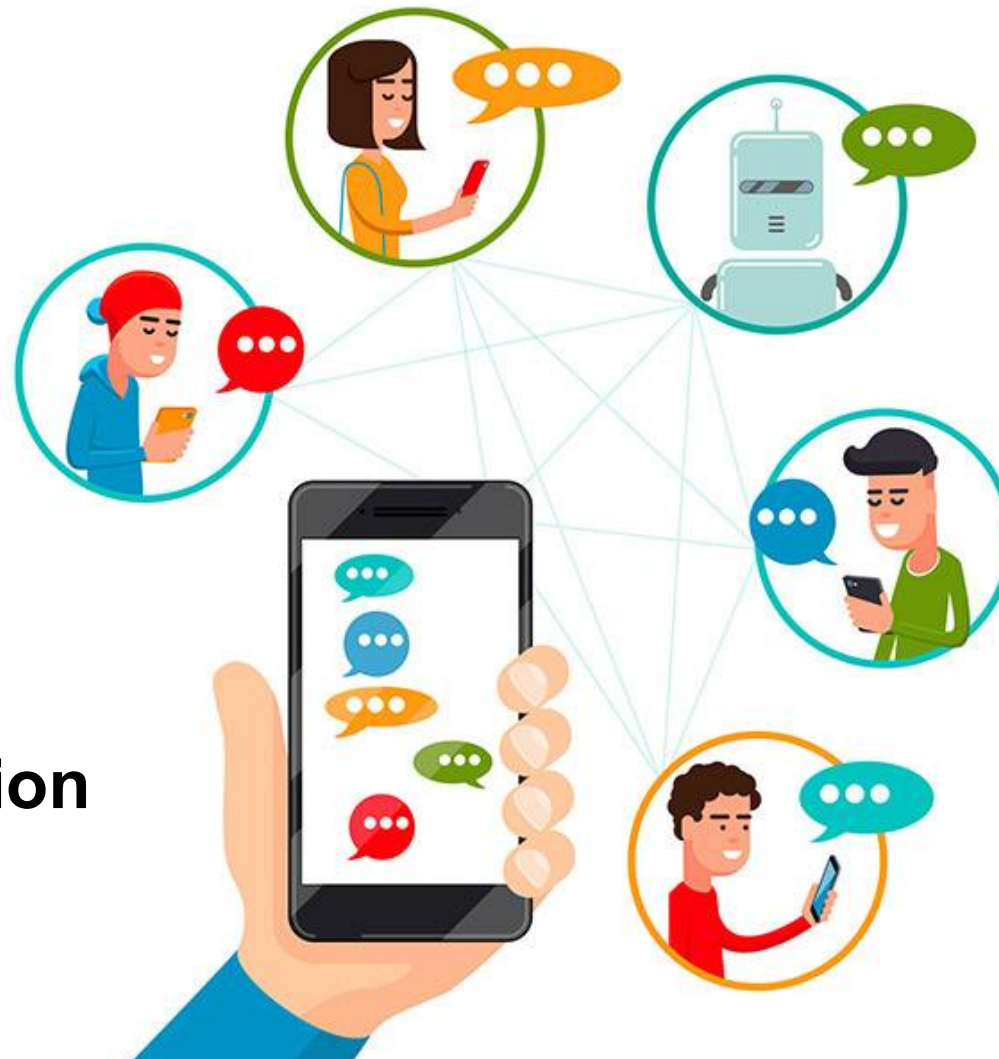
© <https://blogs.sap.com>

- Synchron  $\leftrightarrow$  Asynchrone Methodenaufrufe
  - Vor- und Nachteile
- Asynchrone Programmiermodelle
  - Shared Memory  
(mit Code/Daten-Sperrung)
  - Futures / Promises
  - Callbacks
- Realisierung in anderen Programmiersprachen

- **Gonzales (2017)**  
Mastering Concurrency Programming with Java 9
- **Cleary (2019)**  
Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming
- **Williams (2019)**  
C++ Concurrency in Action
- **Hunter, English (2021)**  
Multithreaded Javascript: Concurrency Beyond the Event Loop
- **Fowler (2022)**  
Python Concurrency With Asyncio

# Zum Schluss ...

**Chatten ist  
Asynchrone  
Kommunikation**



© <https://stitch-ai.com/>