# Asynchrone Programmierung verstehen und anwenden

Eines der grundlegenden Prinzipien bei der Entwicklung in JavaScript ist die asynchrone Programmierung. Sei es bei der Formulierung von Ajax-Anfragen oder beim Lesen von Dateien unter Node.js. In vielen Fällen ruft man eine Funktion auf und übergibt dabei als Parameter ebenfalls Funktionen, die genau dann aufgerufen werden, wenn das Ergebnis der aufgerufenen Funktion feststeht. Zum Beispiel wenn die angefragten Daten vom Server heruntergeladen wurden oder der Inhalt der Datei eingelesen wurde. Der Ablauf des Programms ist dann nicht synchron, sondern asynchron.

Im weiteren Verlauf bekommen Sie anhand von Informationen und Code-Beispielen die sogenannte Callbacks, Promises und async/await erklärt. Callbacks und Promisses sind dabei Hinführung für die Verwendung von async/await, welches am Ende entscheidend ist. Gerne können Sie neben oder anstatt der Texte auch das ca. 30-minütige Video des Youtubers developedbyed anschauen. Hierbei werden die Zusammenhänge recht gut und anschaulich mit Beispielen erklärt.

| **Erklärvideo (optional als Ergänzung)**<br><br>https://t1p.de/l7232 | |
|---|---|

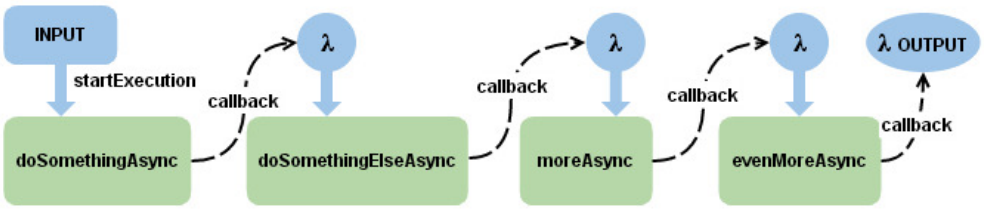
## Das Callback-Entwurfsmuster

Das Übergeben einer Funktion als Parameter einer anderen Funktion, bei der diese übergebene Funktion zu einem späteren Zeitpunkt aufgerufen wird, nennt man Callback-Entwurfsmuster. Der generelle Aufbau folgt dabei folgendem Muster:

**Genereller Aufbau des Callback-Patterns in JavaScript**

```
function asyncFunction(callbackFunction) {
    console.log('Before Callback);
    callbackFunction();
    console.log('After Callback');
}

function callbackFunction() {
    console.log('I am a Callback);
}

asyncFunction(callbackFunction);
```

Die Verwendung von Callback-Funktionen ist ein wesentliches Merkmal der JavaScript-Programmierung. Allerdings kann der übermäßige Gebrauch von Callback-Funktionen zu einem Codegebilde führen, das unter JavaScript-Entwicklern als Pyramid of Doom oder Callback-Hell bekannt ist. Dieses Codegebilde tritt dann auf, wenn asynchrone Funktionsaufrufe übertrieben oft geschachtelt werden.



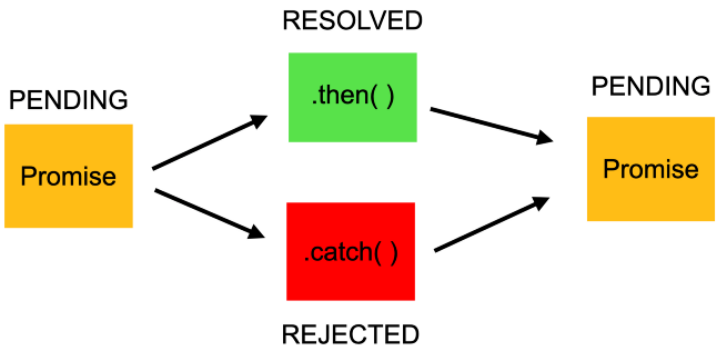

*https://blog.4psa.com/the-callback-syndr 1*

## Die Verwendung von Promises

Promises können die Callback-Hell verhindern. Promises selbst sind nichts anderes als Objekte (vom Typ Promises), die als Platzhalter für das Ergebnis einer asynchronen Funktion dienen. Statt dass die jeweilige asynchrone Funktion selbst die Callback-Handler übergeben bekommt, liefert sie ein Promise-Objekt zurück, das Zugriff auf 2 gekapselte Callback-Funktionen hat: eine, um über den Ergebniswert zu informieren, und eine, um über Fehler zu informieren. In der Regel benennt man diese Callback-Funktion resolve() und reject().

**Code-Beispiel**

```
let myPromise = new Promise(function(resolve, reject) {
    let x = 0;

    if (x == 0) {
      resolve("OK");
    } else {
      reject("Error");
    }
  });

  myPromise.then(
    function successValue(result) {
      console.log(result);
    },
  )

  .catch(
    function errorValue(result) {
          console.log(result);
    }
);
```

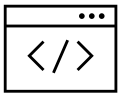



*https://medium.com/analytics-vidhya/java 1*

## Async/Await

Promises vereinfachen das Schreiben von asynchronen Code gegenüber Callback-Funktionen erheblich. Doch es geht noch besser! In Version ES2016 wurde mit den sogenannten *Async Funktionen* eine zusätzliche Möglichkeit eingeführt, die das Formulieren von asynchronem Code noch weiter vereinfacht. Über das Schlüsselwort *async* können dabei asynchrone Funktionen als solche gekennzeichnet werden, wobei das Schlüsselwort vor die Deklaration der entsprechenden Funktion geschrieben wird:

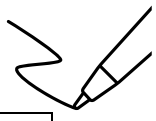
**Code-Beispiel**



```
// First async Function
async function loadFirst() {
      const response = await
      fetch('https://jsonplaceholder.typicode.com/posts/1')
      .then((response) => response.json());

        // simulation of Delay
    setTimeout(() => {
        console.log(response)
    }, 500);
}

// Second async Function
async function loadSecond() {
    const response = await
fetch('https://jsonplaceholder.typicode.com/posts/2')
    .then((response) => response.json());
    console.log(response);

}

loadFirst();
loadSecond();
```

Die setTimeout-Methode im Beispiel dient der Simulation einer Verzögerung, um die Asynchronität besser zu verdeutlichen.

### Arbeitsauftrag



Welche Funktion wird als erstes ausgeben und warum?

*Quelle: Ackermann, Philip: JavaScript – Das umfassende Handbuch, 3. Auflage, 2021 Rheinwerk, S. 849ff.*