

# Entwicklung Web-basierter Anwendungen

Prof. Dr. Stefan Linus Zander

Einführung in JavaScript | Wichtige Konzepte

# Outline

- Error Handling using `try...catch`
- ES6 Syntax
- Closures
- Asynchronous Programming
  - Callbacks
  - Promises
  - Async & Await

# Error Handling

- **Programm defensively**  $\rightsquigarrow$  anticipate the things that can go wrong and handle them in your code
  - e.g. Checking parameter types
  - e.g. Checking for `null` or `undefined`
- Surround code that can throw errors with `try / catch`
  - e.g. in node.js, an unhandled error might cause your server to shut down
- Throw `Error` objects in case of unexpected events

## Error Handling is a kind of mindset

Prepare your code for things that can go wrong & handle them nicely!

```
// NO Error Handling
const printFirstTwoLetters = (str) => {
  const firstTwo = str.substring(0,2);
  console.log(firstTwo); // will not be executed
}
printFirstTwoLetters(5) // str.substring is not a function
```

```
// WITH Error Handling
const printFirstTwoLetters = (str) => {
  "use strict";
  try {
    if (typeof str !== "string")
      throw new Error("Parameter is not a String");
    if (str.length < 2)
      throw new Error("String is less than 2 chars");
    const firstTwo = str.substring(0,2);
    console.log(firstTwo);
  } catch (err) {
    console.log(err)
  }
}
printFirstTwoLetters(5) // Parameter is not a String
printFirstTwoLetters("5") // String is less than 2 chars
```

# Some new ES6 Syntax Features

## Destructuring

```
let robotA = { name: "Bender" };
let robotB = { name: "Flexo" };

let { name: nameA } = robotA;
let { name: nameB } = robotB;

console.log(nameA); // "Bender"
console.log(nameB); // "Flexo"
```

## The Spread Operator

```
let dateFields = [1970, 0, 1];
let d = new Date(...dateFields);
```

## Literal Strings

```
const a = 101;
const b = 42;
const quiz = "Sum of " + a + " + " + b +
  " is " + (a + b) + ".";
```

```
let options = { title: "Menu",
                width: 100,
                height: 200
              };

let {title, width, height} = options;

console.log(title, width, height);
```

```
let obj1 = { foo: 'bar', x: 42 };
let obj2 = { foo: 'baz', y: 13 };
let mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

```
// written as template literal
const quiz =
  `Sum of ${a} + ${b} is ${(a + b)}.`;
```

Source: <https://www.youtube.com/watch?v=a00NRSFgHsY> and <https://javascript.info/destructuring-assignment>

# Closures

- A closure is the combination of **outer** and **inner functions**
- A closure provides access to an outer function's scope from an inner function
- Closures are a common way to achieve **encapsulation**, ie. hiding data from external and uncontrolled access
- To use a closure, define a function inside another function and expose it. To expose a function, return it or pass it to another function.
- The inner function will have access to the lexical scope of the outer function, even after the outer function has returned.
- Usage scenarios
  - Isolation of protected variables
  - Transportation of states to another scope
  - Creation of stateful functions

```
function MyProtectedObj(param) {  
  const mySecretVariable = Math.floor(4711 * Math.random());  
  let name = param;  
  
  return {  
    getCode: function() {  
      return mySecretVariable;  
    },  
    setName: function(value) {  
      name = value;  
    },  
    getName: function() {  
      return name;  
    }  
  }  
}
```

```
let obj = MyProtectedObj("James");  
  
console.log(mySecretVariable);    // Reference Error  
console.log(obj.mySecretVariable); // outputs 'undefined'  
  
obj.getCode();                   //returns the randomly generated number  
obj.setName("John");             //ok  
obj.getName();                   //outputs 'John'
```

# Closure – Pitfalls

Be careful, `this` does not work in closures

```
function MyProtectedObj(name) {  
  this.mySecretVariable = Math.floor(4711 * Math.random());  
  this.name = name;  
  
  return {  
    getCode: function() {  
      return mySecretVariable;  
    },  
    setCode: function(value) {  
      mySecretVariable = value;  
    },  
    getName: function() {  
      return name;  
    }  
  }  
}  
  
let obj = MyProtectedObj("James");  
  
//works since mySecretVariable is bound to the global scope  
console.log(mySecretVariable); // outputs the generated number
```

Sources: <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36> and  
<https://www.computerbase.de/forum/threads/warum-sind-closures-so-wichtig.1906523/>

# Asynchronous JavaScript

# JavaScript is a Single-Threaded, Non-Blocking, Asynchronous PL

## Function Execution Stack (aka **Call Stack**)

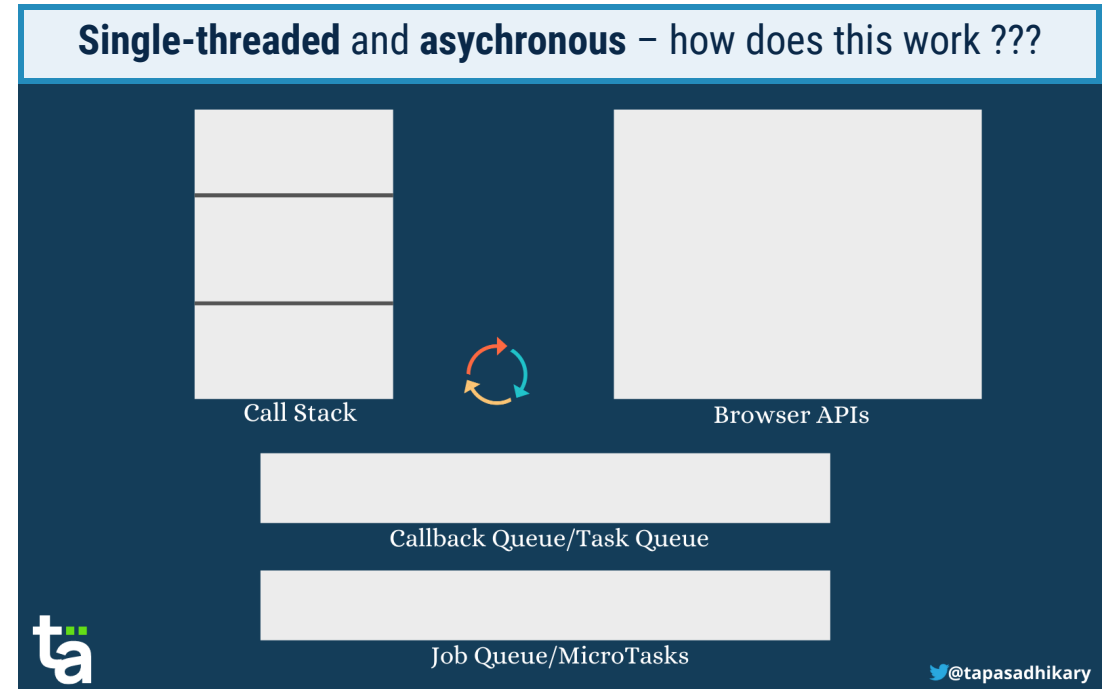
- All invoked functions are added to the call stack
- Completed functions are removed until the stack is empty
- Functions are executed **synchronously** one-by-one

## Callback Queue (aka **Task Queue**)

- Callbacks are stored in this separate (FIFO) queue
- The JS engine **periodically** looks for new entries in the **task queue** and once the **call stack** is empty it shifts the first entry to the call stack and executes it synchronously ( $\Rightarrow$  **event loop**)

## Job Queue

- **Promise executor functions** are stored in the **job queue**
- For each loop of the event loop, one macro task is completed out of the callback queue
- Once that task is complete, the event loop visits the job queue and completes all micro-tasks in the job queue before it continues.



Source: <https://www.freecodecamp.org/news/synchronous-vs-asynchronous-in-javascript/>



# Asynchronous Programming

- JavaScript has some **unique features** for the **asynchronous execution** of code
- The 3 **most important concepts** are

- 1.) Callbacks**
- 2.) Promises**
- 3.) Async & Await**

# Callbacks

- **Callbacks** are a **central element** in **asynchronous JavaScript**
- Callbacks are **(mostly anonymous) functions** that will be called when a previously defined **event** occurs
- Callbacks are implement **handler functions**; they are called **asynchronously** by the JavaScript engine
- Callbacks are most commonly used to ...
  - handle **input events**
  - processes recieved **JSON data** from AJAX requests
- Callbacks can become **problematic** → **Callback-Hell**

```
//***** Example #1: A Simple Callback *****
console.log("Hallo Welt - jetzt");

setTimeout(() => {
    console.log("Hallo Welt - nach 1 Sek.")
}, 1000 );

console.log(
    "Dieser Code wird vor dem asynchronen Code ausgeführt...");

// Output:
// "Hallo Welt - jetzt"
// "Dieser Code wird vor dem asynchronen Code ausgeführt..."
// "Hallo Welt - nach 1 Sek."

//***** Example #2: Event Handler for DOM Elements *****
const btn = document.querySelector('#btn');
btn.addEventListener("click", () => { ... });
```

# Callback-Hell

The **callback-hell** denotes a **christmas tree like pattern** of **nested callback handlers**

```
let i = 0;
let stop = false;

setTimeout(() => {
  console.log("rot - " + i);
  setTimeout(() => {
    console.log("gelb - " + i);
    setTimeout(() => {
      console.log("grün - " + i);
      stop = true;
    }, 2000);
  }, 2000);
}, 2000);

const inc = setInterval(() => {
  i = i + 1;
  if (stop === true) {
    clearInterval(inc);
  }
}, 500);
```

## Output

```
// rot - 3
// gelb - 7
// grün - 11
```

# Promises

- Promises are objects for making **asynchronous calls**
  - a **value** is created in a success case
  - an **error** is created if the promise does not complete
- The **promise constructor** expects an **executor function** with two **callback functions** as arguments
  - `resolve` indicates a **successful completion** of the task
  - `reject` indicates the occurrence of an **error**
- Callback functions are used to announce the **outcome**
- The callback functions are provided by JavaScript
- Promises have three **handler methods**
  - `.then()` accepts `result` and `error` as arguments
  - `.catch()` used to handle error cases
  - `.finally()` used to perform cleanup work

```
// Example
const myPromise = new Promise((resolve, reject) => {
  const rand = Math.floor(Math.random() * 2); // '0' or '1'
  if (rand === 0) {
    resolve(rand);
  } else {
    reject(new Error("Fehlerfall - " + rand));
  }
});

// Promises can be chained instead of nested
// ie., no christmas-tree-pattern
myPromise
  .then((rand) => console.log("Success - " + rand))
  .then(() => console.log("2. Ausgabe nur im Erfolgsfall"))
  .catch((err) => console.error(err));
```

Source: <https://blog.greenroots.info/javascript-promises-explain-like-i-am-five>

# Fetch with Promises

- The **Fetch API** interface allows web browser to make asynchronous HTTP requests without `XMLHttpRequest`
- The `fetch()` method allows to fetch resources asynchronously
  - it takes at least **one argument**: the URL to fetch
  - it does not directly return the data but a **promise** that resolves with a `Response` object
- The `Response` object contains the entire **HTTP response**
  - `.json()` needs to be called to **retrieve** the **JSON data**
- The promise object returned won't be rejected in case of HTTP status codes `404` or `500`

```
const fetch = require('node-fetch'); // not needed in browser

fetch("https://randomuser.me/api/")
  .then((response) => response.json())
  .then((data) => console.log(data.results) )
  .catch((err) => console.error(err));

// Output
[
  {
    gender: 'male',
    name: { title: 'Mr', first: 'Maël', last: 'Da Silva' },
    location: {
      street: [Object],
      city: 'Poitiers',
      state: 'Guadeloupe',
      country: 'France',
      postcode: 54475,
      coordinates: [Object],
      timezone: [Object]
    },
    [...]
  }
]
```

Sources: [https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

# Async & Await – The Preferred Way to handle Promises and Asynchronous Operations

- Async & Await work on top of **promises**
- We use `async` to **return a promise**<sup>1</sup>
  - `async` declares an asynchronous function
  - transforms a function into a `Promise`
  - enable the use of `await`
  - resolve with whatever is returned by its body
- We use `await` to wait and **handle a promise**
  - `await` pauses the execution of asynchronous functions
    - until a promise is settled (either resolved or rejected)
    - and a value/error is returned/thrown
  - `await` is used in front on promises
  - only works with promises, not callbacks
  - can only be used inside `async` functions

```
const fetchUserWithErrorHandling = async () => {
  try {
    const res = await fetch(url);
    const data = await res.json();
    console.log("finished");
  } catch(err) {
    console.error(err);
  }
}

fetchUserWithErrorHandling();
```

- If the promise rejects, it throws an error that is handled by the `catch` block
- Async/Await enables traditional error handling with `try...catch`

<sup>1</sup> If you do not return a promise explicitly from an async function, JavaScript automatically wraps the value in a Promise and returns it.

# Await must be Invoked in an async Function

```
const fetchUserDetails = async (userId) => {  
  // pretend we make an asynchronous call  
  // and return the user details  
  return {'name': 'Robin', 'likes': ['toys', 'pizzas']};  
}
```

```
// not working  
const user = await fetchUserDetails();  
console.log(user);
```

```
// correct solution via IIFE  
(async () => {  
  const user = await fetchUserDetails();  
  console.log(user);  
})();
```

## Remember

An async function always encapsulates its return value in a promise

## Remember

`await` can only be called inside an async function

## Remember

In order to use `await` regardless of an async function, it needs to be wrapped in an async IIFE

Source: <https://blog.greenroots.info/javascript-async-and-await-in-plain-english-please>